MorphingDB: A Task-Centric AI-Native DBMS for Model Management and Inference

Sai Wu
Zhejiang University
China
wusai@zju.edu.cn

Ruichen Xia
Zhejiang University
China
rcxia@zju.edu.cn

Dingyu Yang
Zhejiang University
China
yangdingyu@zju.edu.cn

Rui Wang
Zhejiang University
China
rwang21@zju.edu.cn

Huihang Lai
Institute of Computing Innovation,
Zhejiang University
China

Jiarui Guan
Jiameng Bai
Zhejiang University
China

Dongxiang Zhang
Xiu Tang
Zhejiang University
China

Zhongle Xie
Peng Lu
Zhejiang University
China

Gang Chen
cg@zju.edu.cn
Zhejiang University
China

Abstract

The increasing demand for deep neural inference within database environments has driven the emergence of AI-native DBMSs. However, existing solutions either rely on model-centric designs requiring developers to manually select, configure, and maintain models, resulting in high development overhead, or adopt task-centric AutoML approaches with high computational costs and poor DBMS integration. We present MorphingDB, a task-centric AI-native DBMS that automates model storage, selection, and inference within PostgreSQL. To enable flexible, I/O-efficient storage of deep learning models, we first introduce specialized schemas and multi-dimensional tensor data types to support BLOB-based all-in-one and decoupled model storage. Then we design a transfer learning framework for model selection in two phases, which builds a transferability subspace via offline embedding of historical tasks and employs online projection through feature-aware mapping for real-time tasks. To further optimize inference throughput, we propose pre-embedding with vectoring sharing to eliminate redundant computations and DAG-based batch pipelines with cost-aware scheduling to minimize the inference time. Implemented as a PostgreSQL extension with LibTorch, MorphingDB outperforms AI-native DBMSs (EvaDB, Madlib, GaussML) and AutoML platforms (AutoGluon, AutoKeras, AutoSklearn) across nine public datasets, encompassing series, NLP, and image tasks. Our evaluation demonstrates a robust balance among accuracy, resource consumption, and time cost in model selection and significant gains in throughput and resource efficiency.

SIGMOD '26, Bengaluru, India

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3769844

CCS Concepts

• **Information systems** → **Data management systems**; • **Computing methodologies** → **Machine learning**.

Keywords

AI-native DBMS, Task-centric, Model selection

ACM Reference Format:
Sai Wu, Ruichen Xia, Dingyu Yang, Rui Wang, Huihang Lai, Jiarui Guan, Jiameng Bai, Dongxiang Zhang, Xiu Tang, Zhongle Xie, Peng Lu, and Gang Chen. 2025. MorphingDB: A Task-Centric AI-Native DBMS for Model Management and Inference. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD '26).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3769844

1 Introduction

AI-native DBMSs integrate deep learning inference into database engines to support advanced cognitive tasks such as face detection [51], object recognition [57], and entity resolution [20]. These systems extend traditional relational and vector databases, including PostgreSQL [13], Vexless [48], VectorDB [54], and AquaPipe [56], to provide unified data and model processing capabilities. Existing implementations typically follow a **model-centric** paradigm, obligating engineers to oversee the entire model lifecycle from architecture design and parameter tuning to deployment and version control [38]. This approach demands deep machine learning expertise and manual configuration of complex model details, resulting in a substantial increase in **development overhead**.

In contrast, several modern automated machine learning (AutoML) frameworks [22, 30, 32, 50] adopt **a task-centric paradigm** without requiring deep expertise in data science or machine learning. By iteratively sampling, training, and evaluating a pool of candidate models, these systems automatically identify the optimal model for any dataset and task. Motivated by this philosophy,

we embed **the task-centric module** into relational databases, enabling data engineers to develop complex AI applications via familiar SQL interfaces without manual model store, model selection, and hyperparameter tuning.

Table 1 provides a comparative analysis of the model-centric and task-centric paradigms, highlighting differences in operator sets, task definitions, and query interfaces. In the **model-centric** paradigm, executing the same analytical task across multiple datasets requires developers to explicitly **manage model storage and selection**. Each model must be registered with specific metadata, such as model name, version, input/output schema, and invocation interface. Data developers are responsible for identifying suitable models for different data modalities, performing parameter tuning, and integrating these models into the query logic. For example, one might deploy a *U-Net* [43] model to segment iPhone 16 product images, whereas *Mask R-CNN* [27] is employed to detect and classify user comments in iPhone 16 reviews, which heavily rely on domain expertise.

In contrast, the **task-centric** paradigm abstracts away low-level model details through high-level task declarations such as *ImageRecognition* or *SentimentClassification*. Instead of requiring developers to explicitly specify which model to use, the system automatically resolves the task to the most appropriate model based on data characteristics, system context, and available resources. This eliminates the need for manual model selection and fine-tuning, significantly reducing the barrier to entry for non-expert users. Task-centric systems streamline AI integration workflows and scale more efficiently across diverse use cases.

While the task-centric paradigm offers significant abstraction and usability benefits, directly adopting it within relational databases introduces **some challenges**:

- Model Storage: Relational databases such as PostgreSQL are designed for structured data and lack native support for unstructured model components such as neural network architectures. While extensions like *pgvector* allow storage of flat vectors, they omit essential tensor shape metadata, making them incompatible with PyTorch tensors. Alternatively, API-based remote models offer convenient access to powerful closed-source LLMs without local deployment, but introduce challenges including external dependencies, unpredictable latency, and rate limitations.
- Model Selection: Traditional AutoML pipelines determine the most suitable model for a given task and dataset through **iterative training and evaluation of multiple candidates**. While effective, this process is **computationally expensive** and poorly suited for online inference scenarios that demand low latency and real-time responsiveness.
- Query Efficiency: When inference operators are interleaved with relational operators (e.g., joins, filters, aggregations), conventional query engines often fail to **optimize on these heterogeneous operations**. They lack awareness of **model execution costs**, **hardware-specific constraints** (e.g., GPU availability and batching), and **data dependencies** introduced by model invocation. These limitations hinder the system's ability to achieve high-throughput inference at scale—especially in task-centric workloads where model inference is repeatedly applied over large datasets.

To address the above challenges, we build the system MorphingDB to integrate the task-centric paradigm in DBMS and bridge the performance gap between DBMS and deep learning frameworks.

Firstly, we design specialized model schemas and data types tailored for storing and managing deep learning models within relational databases. We introduce the **Mvec** representation, which efficiently encodes multi-dimensional tensors with explicit shape information to enable lossless conversion and in-database manipulation with PyTorch tensors. In addition to BLOB-based storage, MorphingDB implements a **decoupled architecture model storage**, which separates model structure (e.g., network layers and operations) from its trained weights. This design facilitates partial loading and fine-grained inference execution, enhancing scalability and resource efficiency in model management.

Secondly, we define the model selection problem as a transfer learning task, employing a two-phase framework consisting of an offline embedding phase and an online model selection phase. The core idea is to build a transferability-related subspace, mapping the models and tasks into this subspace. Within this subspace, the distance between model vectors and task vectors signifies the level of transferability. During the offline phase, we extract transfer patterns from performance matrices of candidate models across diverse historical tasks. In the online phase, we develop a projection mechanism that aligns target tasks into the same subspace through feature-aware mapping. This two-phase framework enables real-time model selection via efficient vector multiplication without extensive fine-tuning, saving resources and time overhead.

Finally, MorphingDB introduces two core optimization techniques that enhance tensor manipulation and inference efficiency. (1) It employs an innovative approach that combines pre-embedding with SIMD to share vectorized data, thereby eliminating redundant embedding computations. (2) To maximize device utilization and scalability, MorphingDB implements a pipeline-based batch inference strategy using a Directed Acyclic Graph (DAG) of operators. By leveraging adaptive CPU-GPU device placement and applying advanced pipeline techniques to optimize execution order, MorphingDB streamlines complex inference workflows and minimizes bottlenecks.

We conduct a comprehensive comparative analysis against several prominent systems, EvaDB [8], Madlib [7], and GaussML [25] on nine public datasets, encompassing series, NLP, and image tasks. We also utilize the CIFAR dataset to evaluate the multi-dimensional performance of model selection in multiple popular AutoML frameworks such as AutoGluon [50], AutoKeras [30], and AutoSklearn [22]. Our results indicate MorphingDB consistently outperforms other approaches in both performance and usability, offering an intuitive interface that simplifies AI task execution. Compared to the AutoML framework, it has a better balance between accuracy, resource efficiency, and training time in model selection.

The paper is organized as follows: Section 2 presents the system overview. Sections 3–5 cover model storage, selection, and optimization. Section 6 reports experiments, followed by related work in Section 7 and conclusion in Section 8.

Table 1: Two Design Paradigms

Features	Model-Centric	Task-Centric
Operators	Create Model, Drop Model, Predict Model, Update Model, Prune Model, ...	Select Model for Task, Register Task, Predict Task
Definition	USING engine = 'mlflow', model_name = 'folder_name', mlflow_server_url = 'http://0.0.0.0:5001/'	CREATE TASK sentiment_classifier (INPUT=Text_Blob, OUTPUT in 'POS, NEG, NEU', Type ='Classification')
Query Interface	**SELECT** U.gender, U.location, AVG(*TextCNN* ForSentiAnalysisV_2_0(R.comment)) **FROM** user **AS** U, Product **AS** P, review **AS** R **WHERE** U.ID=R.uid **AND** P.id=R.pid **AND** *UNet*ForImageRecognitionV_1_2(P.img)= "iPhone 16" **AND** *MaskRCNN*ForImage- RecoginitionV_0_2(R.img)="iPhone 16" **GROUP BY** U.gender, U.location	**SELECT** U.gender, U.location, AVG(sentiment_classifier(R.comment)) **FROM** user **AS** U, Product **AS** P, review **AS** R **WHERE** U.ID=R.uid **AND** P.id=R.pid **AND** ImageRecognition(P.img)="iPhone 16" **AND** ImageRecognition(R.img)="iPhone 16" **GROUP BY** U.gender, U.location

2 System Overview

2.1 Task-Centric Paradigm

MorphingDB introduces a *task-centric* paradigm to simplify the incorporation of deep neural models within database-driven analytical dataflows. Rather than requiring developers to explicitly construct and manage models, which is typical in model-centric systems, MorphingDB enables users to define high-level inference tasks (e.g., face detection or sentiment analysis), allowing the system to handle model selection, configuration, and deployment automatically. This shift from model-centric control to task-centric specification not only reduces the barrier to entry for non-expert users, but also enables declarative, resource-efficient deployment of AI functionality directly within the DBMS.

Let T denote the set of tasks that users aim to accomplish with their data, and let M represent the set of models available in our repository. The task-centric paradigm is formalized as a function $f : T \rightarrow M$ that selects the most appropriate model from M for each task in T based on performance and accuracy metrics.

By allowing users to specify what task to perform rather than *how to perform it*, MorphingDB aligns with the declarative nature of traditional DBMSs, where users define desired outcomes rather than detailed execution procedures. This abstraction significantly reduces development complexity and improves system adaptability, enabling MorphingDB to more easily evolve with changing data workloads and analytical needs.

2.2 MorphingDB Overview

The whole system is designed with a three-layer architecture to ensure efficient model management and inference execution. The storage layer handles the persistence of models and their associated metadata, enabling prompt retrieval and structured organization. The processing layer is responsible for orchestrating model selection, loading, and batch-based execution, optimizing computational efficiency and resource utilization. The interface layer provides an SQL-based interface for task inference, allowing users to integrate machine learning capabilities into database workflows. A detailed overview of MorphingDB's primary workflow is illustrated in Figure 1.

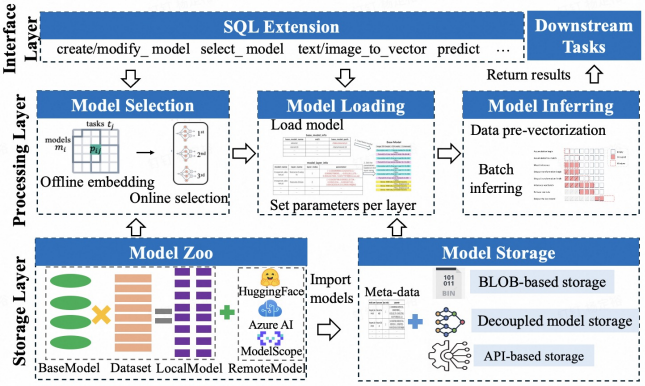


Figure 1: Overview of MorphingDB

Within the storage layer, MorphingDB maintains a unified **model zoo** encompassing both locally stored pre-trained models and externally hosted models, along with diverse datasets. Users can easily import models—either local or remote—into the system without requiring additional configuration. MorphingDB automatically records model metadata in a centralized repository table. To accommodate varying storage requirements, MorphingDB supports multiple model formats: (1) **BLOB-based storage**, where both model architecture and parameters are stored as a single serialized object, (2) **Decoupled storage**, which separates model structure from parameters to enable selective loading, fine-grained updates, and better runtime flexibility. (3) **API-based model integration**, where external endpoints can be registered as logical operators. This allows remote models to be seamlessly embedded into SQL queries alongside local models.

In the processing layer, MorphingDB initiates the model selection process to identify the most appropriate pre-trained model from the model zoo. This process comprises offline task-model embedding to capture task-model relationships, followed by online model selection to dynamically choose the best model based on task requirements and performance metrics. Once a model is selected, MorphingDB loads it by retrieving the model metadata and progressively configuring layer-wise weight and bias parameters.

Following this, our system executes the model inference process and generates results for downstream tasks. Notably, throughout the designed pipeline, data vectorization for inference and batch inference optimizations are implemented to enhance inference efficiency.

The interface layer provides an intuitive and structured approach for users to define machine learning tasks through an extended SQL syntax. To facilitate custom ML task definitions, this layer allows users to precisely specify input/output formats, expected neural model types, and performance constraints. By integrating these capabilities directly into SQL, MorphingDB eliminates the need for low-level implementation details, making model deployment and inference more accessible.

MorphingDB is implemented in C++ and integrated into PostgreSQL as an extension. The system incorporates LibTorch [11], the C++ version of PyTorch, to perform inference tasks. While a standalone Python library may simplify development, it hinders tight integration with the DBMS query engine, making it difficult to support operator-level scheduling, cost-based optimization, and SQL-native model invocation. Furthermore, Python-based solutions introduce runtime overhead, fragile dependencies, and complex deployments, limiting robustness and scalability in enterprise and cloud-native environments.

Discussion. By abstracting inference as query operators and supporting heterogeneous backends (e.g., GPUs, remote APIs, decoupled models), MorphingDB enables efficient, composable, and context-aware inference over both structured and unstructured data. This design naturally generalizes to LLM-based agentic systems: complex agents can be modularly decomposed into subtasks (e.g., retrieval, reasoning), mapped to LLMs of different capacities, and executed as directed acyclic graphs (DAGs). To optimize such workflows, MorphingDB's cost model can be extended for agent-aware planning, selecting models based on latency–accuracy–resource trade-offs. These capabilities position MorphingDB as a foundation for AI-native agents that demand declarative task specification and fine-grained control over model execution across heterogeneous environments.

3 Inner-DB Model Management

In this section, we present how deep neural models are stored and managed in MorphingDB. Specifically, we introduce a Mvec representation format to facilitate the storage for multi-dimensional tensors.

3.1 Model Storage

We store the trained neural network models in a model repository for subsequent AI tasks and perform model inference computations by creating and loading models. A trained neural network model usually needs to save information such as network architecture, weight parameters, bias parameters, optimizer state, and other hyperparameters. MorphingDB supports local model storage through two mechanisms: the binary large object (BLOB) file storage and the decoupled storage. In addition, it provides API-based integration to support remote or externally hosted models, including proprietary or closed-source services [14].

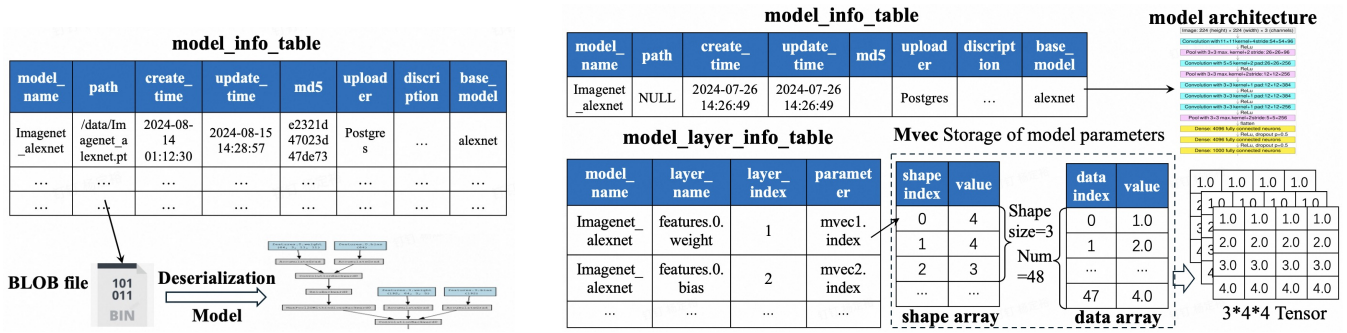
BLOB-based model storage. This strategy serializes both the neural network architecture and its parameters into a single binary large object. Model metadata, such as identifier, version, and storage path, is maintained in a *model_info_table* shown in Figure 2a. Although this design leverages the file system for straightforward implementation, it incurs substantial deserialization overhead during model loading, undermining low-latency inference. Its monolithic format also restricts fine-grained component loading—hindering pipeline parallelism, scalability, and the reuse of shared architectural elements—and disallows partial updates to model parameters. Consequently, MorphingDB reserves BLOB-based storage for static models with minimal update or sharing requirements.

Decoupled storage of model architecture and parameters. To enable data reuse, partial model updates, and partial model loading, MorphingDB proposes decoupled model storage, which extracts the model network architecture as the base model, such as ResNet18, ResNet50, AlexNet, etc., from the model weight and bias parameters of different layers, and stores them separately. As illustrated in Figure 2b, we also use the *model_info_table* to record the model metadata, and the *base_model* field to record the pointers to the base architecture. An extra *model_layer_info* table is used to record the information of each model layer, including layer name, layer index, and the weight and bias parameters. Decoupled storage is similar to ONNX-model storage [12, 17], which is an open format focused on model exchanges for various LLM models and different platforms. While storing base model weights and fine-tuned diffs externally in formats such as ONNX or LoRA-style deltas is feasible, it presents practical trade-offs. External storage introduces additional runtime dependencies and latency, and can complicate integration with in-DBMS planning and scheduling, potentially undermining optimization opportunities and system reliability.

API-based model storage. MorphingDB abstracts external model endpoints as logical operators within its task-centric execution framework. Each API-based model is registered with metadata such as endpoint URL, input/output schema, expected latency, authentication credentials, and usage quotas. During query planning, these operators are treated equivalently to local models through HTTP/REST or gRPC interfaces with automatic request construction, payload serialization, and result deserialization. To ensure robustness and performance, MorphingDB includes adaptive retry logic, timeout control, and response caching.

Discussion. MorphingDB abstracts over heterogeneous model integration strategies—ranging from in-database to API-based deployment—within a unified task execution framework. This design allows users to flexibly select the most appropriate integration method based on deployment constraints, workload characteristics, and performance objectives. Some advantages and limitations are summarized as follows:

- BLOB-based storage: This method stores both the model architecture and parameters as a single binary object, offering simplicity and fast loading for small to medium-sized models. It is ideal for static or single-tenant scenarios where models rarely change and update granularity is not required, but lacks reuse flexibility and incurs redundancy across variants.



(a) BLOB-based all-in-one model storage

(b) Decoupled model storage

Figure 2: Model storage and management in MorphingDB

- Decoupled storage: MorphingDB separates architecture metadata from weight tensors, supporting selective loading, fine-grained updates, and partial inference execution. This reduces redundancy across fine-tunes (e.g., ResNet-50 variants) and is well-suited for multi-tenant, memory-constrained, or high-throughput settings.
- API-based integration: This mode enables easy access to powerful, closed-source LLMs without requiring local deployment or fine-tuning, making it ideal for lightweight tasks, exploratory queries, or rapid prototyping. However, it introduces external dependencies, variable latency, and limited optimization within the query planner.

3.2 Mvec Tensor Representation

While formats like ONNX are widely used for model representation and exchange, they are not optimized for fine-grained, database-native storage and access patterns. To enable in-database storage of multi-dimensional model parameters and direct interoperability with PyTorch, we introduce a new tensor format-**Mvec representation**. Unlike PostgreSQL's *pgvector* extension, which only accommodates one-dimensional vectors by storing raw values and vector length without shape metadata, Mvec preserves both the tensor data and its full dimensionality, enabling accurate, lossless conversions between database-resident tensors and PyTorch tensors. Each tensor in Mvec is represented by two contiguous arrays (Figure 2b):

- A **shape array**, which records the size of each tensor dimension (e.g., channels, height, width);
- A **data array**, that stores the tensor's elements as a flattened, one-dimensional sequence in row-major order, meaning that elements in the same row are stored contiguously before moving to the next row.

When ingesting a PyTorch tensor, MorphingDB serializes its dimensions into the shape array and its values into the data array. Conversely, to reconstruct the original tensor, the system reads the shape array to determine dimensionality, computes stride information to map one-dimensional indices back to multi-dimensional coordinates, and reshapes the data array accordingly. MVec is designed specifically to support shape-aware binary tensor storage tightly integrated with the DBMS, enabling efficient SQL-level filtering, slicing, and partial loading of tensor data—operations that ONNX does not natively support. By mirroring PyTorch's contiguous memory layout, Mvec ensures efficient storage, rapid I/O, and seamless tensor reconstruction within the DBMS.

3.3 Model Zoo

To support heterogeneous task requirements across diverse data modalities, MorphingDB incorporates a modular and extensible **Model Zoo**, which serves as the backbone for automated, task-driven model selection. This repository is populated with a rich collection of pretrained models sourced from established model hubs such as the PARC benchmark [19], Hugging Face [15], and Ollama [16], enabling out-of-the-box support for a wide range of AI tasks without requiring users to collect datasets, fine-tune models, or procure GPU resources. For instance, ResNet variants [26] such as ResNet-18/34 are optimized for edge scenarios like camera-based defect detection, while ResNet-50/101 serve higher-accuracy tasks like facial recognition and object tracking. Similarly, the YOLO family [23] offers a spectrum from lightweight versions (e.g., YOLOv5-nano) for mobile inference to larger variants (e.g., YOLOv5-x, YOLOv8) for industrial-grade detection. These variants differ in architecture, fine-tuning data, and deployment profiles, underscoring the need for systems like MorphingDB to manage diverse model variants efficiently and contextually.

The Model Zoo is organized by task type and data modality, supporting image classification, time series analysis, and natural language processing, each with distinct input characteristics, data distributions, and semantic structures. For image tasks, the Zoo includes tiered convolutional architectures from the PARC benchmark [19], such as ResNet-50, ResNet-18, and AlexNet, each pretrained on canonical datasets like ImageNet or CIFAR-10 to ensure broad generalization. For time series applications, we integrate lightweight MLP-based models trained across datasets with varying temporal granularity [21], supporting regression and classification objectives. In the NLP domain, we include transformer-based encoders such as ALBERT [37] and RoBERTa [45], fine-tuned for tasks such as domain-specific sentence classification and semantic matching.

Unlike conventional model-centric frameworks that require explicit user configuration of architectures and hyperparameters, MorphingDB adopts a task-centric interface: model selection is guided by high-level task semantics and dataset properties. The Model Zoo thus functions not only as a storage layer for local pretrained models or remote API-based models, but also as a substrate for automated matching during inference planning, enabling efficient, domain-adaptive deployment without user intervention.

Currently, the Model Zoo contains 198 models across the three primary modalities (image, time series, text), and is designed to support future extensions through plug-in architectures and continual ingestion of models. This design ensures that MorphingDB remains adaptable to evolving workloads, providing a robust foundation for unified, multi-task AI computation within relational databases.

4 Model Selection

To address the high cost of iterative training and evaluation across multiple models and datasets, MorphingDB employs a two-phase model selection strategy. First, an offline embedding phase constructs a transferability-aware subspace that captures relationships between tasks and models. Then, during the online phase, the system performs real-time model selection through lightweight vector operations without requiring feature extraction or fine-tuning, substantially reducing resource consumption and latency.

4.1 Problem definition.

Adopting the most appropriate model for different tasks can essentially be seen as a transfer learning problem. Although the matching score can be estimated using the correlation between the label and the forward features, the time complexity of this approach grows linearly with the number of models. This paper delves into the intricate relationship between tasks and models, introducing a novel methodology to reduce time overhead. The core idea is to build a transferability-related subspace, mapping the models and tasks into the subspace. Within this subspace, the distance between model vectors and task vectors signifies the level of transferability.

Formally speaking, given a target task t^*, our goal is to select a suitable model m^* from the massive model zoo $\mathcal{M} = \{m_i\}_{i=1}^{M}$ so that the model m^* can achieve best performance on t^*. To this end, we compute a transfer score $\text{Trans}(m_i, t^*)$ between each model m_i and the target task t^*, as shown in Equation 1. We solve this problem by offline model/task embeddings and online model selection.

$$m^* = \arg \max_{m_i \in \mathcal{M}} \text{Trans}(m_i, t^*). \tag{1}$$

4.2 Offline: model and task embeddings.

To construct a subspace related to transfer capabilities for mapping models and tasks, we initially collect the real transfer performance of historical tasks $\mathcal{T} = \{t_j\}_{j=1}^{N}$ on the models in the model zoo $\mathcal{M}$ during the offline phase. The historical transfer matrix is denoted as $V \in \mathbb{R}^{M \times N}$, where N is the number of historical tasks. Each element $v_{ij} \in V$ represents the performance of model i on task j. Subsequently, we extract latent transfer patterns representing the performance of models across various historical tasks from the V.

The goal of matrix factorization is to decompose a matrix into multiple low-dimensional matrices, which reside in the k-dimensional subspace. The embeddings in the latent subspace represent the underlying features of the data, capturing significant patterns within it. In our scenario, the k-dimensional subspace can be regarded as latent factors influencing model-to-task transferability. Specifically, we decompose this matrix with the objective function in Equation 2, where $W \in \mathbb{R}^{M \times k}$ and $H \in \mathbb{R}^{N \times k}$ respectively represent the embeddings of models and historical tasks. The inner product of model embedding $\boldsymbol{m_i} \in W$ and task embedding $\boldsymbol{t_j} \in H$ reflects the matching degree between them.

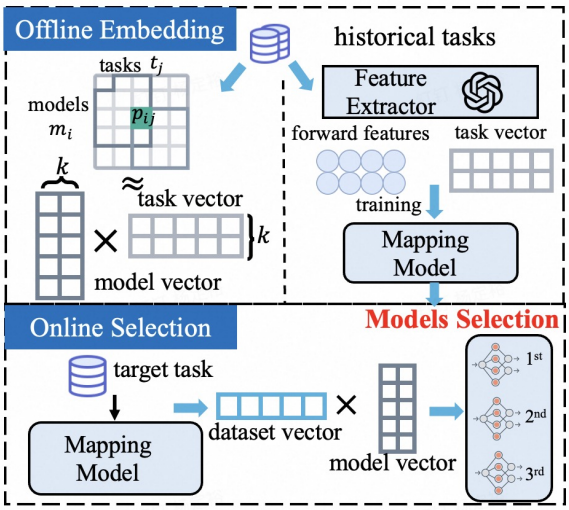


Figure 3: Model selection based on task embeddings

$$\min_{W,H} \|V - WH^\top\|_F^2 \quad \text{subject to } W, H \geq 0 \tag{2}$$

where $\|\cdot\|_F$ denotes the Frobenius norm.

4.3 Online: model selection.

In the online phase, our goal is to select the most suitable model for a target task t^*. Since the embeddings of historical tasks lie in the learned k-dimensional latent space, selecting a model requires the embedding $\boldsymbol{t^*}$ of the new task. However, directly computing this embedding is non-trivial as t^* was not involved during training.

To address this, we leverage the powerful generalization capabilities of Large Vision Models (LVMs), such as CLIP, which are pre-trained on a wide range of vision and text tasks. Specifically, we collect forward features of historical tasks using the LVM and pair them with their learned embeddings $H \in \mathbb{R}^{N \times k}$. Using this data, we train a regression model R to learn the mapping from the LVM feature space to the latent task embedding space. We adopt the CLIP model as the forward feature extractor and a random forest as the regressor for training, as shown in Equation 3. The reason we adopt CLIP is that CLIP supports both image and text modalities, equipping our approach with the ability to perform model selection across different modalities. This approach is grounded in the assumption that tasks exhibiting similar LVM features tend to share similar transferability characteristics, enabling the regressor

to generalize to unseen tasks.

$$\boldsymbol{t_j} = R_{random\_forest}(CLIP.encode\_image(t_j)) \quad (3)$$

The embedding of target task $\boldsymbol{t^*}$ can be deduced by passing its LVM forward features to the regressor R. Finally, the transfer scores $\text{Trans}(m_i, t^*)$ of the models in the repository on the target task can be obtained by multiplying the learned model embeddings with the task embedding. As shown in Equation 4, we can directly rank the estimated transfer scores to select the most suitable model m^*. This model selection method reduces the need for feature extraction and extensive fine-tuning on the target task for each model in the model repository, significantly saving resources and time overhead involved in model selection.

$$m^* = \arg\max_i(\text{Trans}(m_i, t^*)) = \arg\max_i(\boldsymbol{m_i} * \boldsymbol{t^*} \mid \boldsymbol{m_i} \in H) \quad (4)$$

5 Inner-DB inference Optimizations

To bridge the performance gap between DBMS and ML frameworks, MorphingDB incorporates two core optimization techniques designed to enhance tensor manipulation and inference efficiency: optimized vectorized sharing, and pipelined batch inference.

5.1 In-Database Vectorized Sharing

In traditional inference pipelines, each query involving images, text, or other modalities triggers an on-the-fly embedding process, followed by model-specific inference. As a result, even when the same dataset is analyzed repeatedly, the system redundantly computes embeddings for each query, leading to unnecessary overhead and increased latency.

We find that the feature extraction stage—responsible for converting raw data into vectorized embeddings—is decoupled from the subsequent inference stage. Once data is embedded, the resulting vectors are independent of the particular inference model used, which allows them to be effectively reused across diverse downstream tasks. Therefore, we propose a pre-embedding and sharing mechanism to mitigate the inefficiencies caused by repeatedly embedding data for diverse user tasks and inference operations.

The process begins with a dedicated feature extraction pipeline that converts raw data into high-dimensional feature vectors optimized for database storage, as illustrated in Figure 4. For text data, we employ pre-trained language models such as ALBERT [37] or BGE embedding [53] to generate semantic embeddings. For image data, widely adopted convolutional neural network architectures, including ResNet [36], VGG [46], and Inception [49], are used to extract compact vector representations. These embeddings are stored in specialized database tables and organized into vector blocks that support various AI inference tasks.

To further enhance performance, we integrate SIMD (Single Instruction, Multiple Data) instructions into the vectorization process. Rather than processing image pixels sequentially, where each pixel is individually scaled and transformed, our approach processes groups of pixels concurrently. For example, during image normalization, multiple pixel values are loaded simultaneously into SIMD registers, which perform parallel arithmetic operations to apply the necessary transformations across the entire group in a single operation. An analogous strategy is applied to text data, where simultaneous processing of multiple token embeddings accelerates

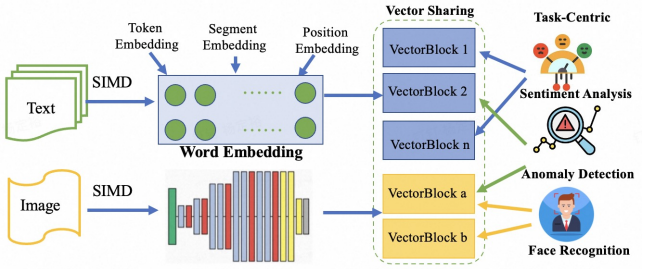


Figure 4: In-Database Vectorization Sharing

the generation of semantic representations. This batch processing strategy minimizes computational overhead for individual queries and enables the precomputation of embeddings that can be efficiently reused across multiple inference tasks, thereby optimizing overall system throughput and scalability.

5.2 Batch Inference in Pipeline

Previous research [13, 55] has seen the development of AI databases within PostgreSQL that cater to various machine-learning tasks. However, these databases often face challenges in inference efficiency, particularly when dealing with intricate queries and extensive datasets–a hurdle also encountered by ML researchers in their early stages. While some efforts have been made to modify the PostgreSQL core, these modifications can pose difficulties during kernel upgrades, limiting the incorporation of community features. To overcome the efficiency shortcomings without the need for core modifications to PostgreSQL, MorphingDB introduces a batch inference pipeline strategy for AI databases. This strategy aims to maximize device utilization and scalability to enhance overall performance.

Figure 5 illustrates an example of our batch inference pipeline framework, which contains multiple relational algebra operators (i.e., *JOIN*, *FILTER*) and model inference function *Predict*. MorphingDB first parses the SQL statement into a series of operators. These operators are structured as a Directed Acyclic Graph (DAG), where each node represents an individual operation (e.g., *SCAN*, *JOIN*, *FILTER*), and the edges denote dependencies between them. If the query has AI inference functions, our inference model is recognized as a specialized node in the DAG and then fetches the model structure, weights, and hyperparameters stored in the database. For example, an Alexnet model has to assemble the parameters of *Conv*, *MaxPooling*, *Flatten*, and *Dense* Layers into an executable inference module. The assembly of inference functions allows the system to adapt to different models and task requirements, enhancing flexibility and scalability in the inference pipeline.

CPU+GPU Co-processing: As shown in Figure 5, one complex SQL query might integrate relational algebra operators and AI model inference functions, within a database system optimized for AI tasks. This requires overcoming the inherent differences in computational characteristics between relational algebra operators (e.g., *JOIN*, *FILTER*, *AGGREGATE*) and AI inference functions (e.g., *Predict*, *SentimentClassifier*). Relational operators typically rely on CPU-bound execution due to their control flow-heavy nature and the need for efficient handling of random memory access

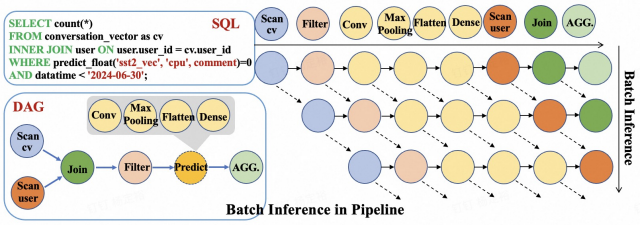


Figure 5: The pipeline processing of batch inference

and branching logic. These operators are generally optimized on multi-core CPUs, where techniques such as multi-threading can accelerate operators like filtering, joining, and aggregating large datasets. Traditional AI inference tasks also perform reasonably well on CPUs, especially for models with lower computational operators. However, when handling more complex deep learning networks, such as convolutional neural networks (CNNs) or transformers, the latency introduced by CPU-bound execution becomes prohibitively high. Our framework supports leveraging GPUs to accelerate these compute-intensive inference operators like large-scale matrix multiplications, convolutions, and attention mechanisms, which require high levels of parallelism and memory bandwidth.

The execution of heterogeneous operators introduces overhead from data movement between CPU and GPU memory spaces, especially when combined with network latency incurred by remote model access. These memory and network transfers can significantly impact end-to-end performance, adding latency that may offset the computational gains of GPU acceleration. Therefore, optimizing execution plans requires careful orchestration to balance GPU throughput with memory synchronization and remote communication costs, ensuring that the benefits of hardware acceleration and model integration justify the overhead incurred.

We propose a cost model to determine the optimal execution device (CPU or GPU) for both traditional relational operators and AI inference operators within an SQL query pipeline. The total cost of an operator op is estimated as:

$$C_{op} = ExecTime_{op} + TransCost_{op} \tag{5}$$

where $ExecTime_{op}$ denotes the operator's local or remote execution time, and $TransCost_{op}$ accounts for data transfer time—either between CPU and GPU memory spaces or across the network when invoking remote models. Notably, for external, closed-source models where $ExecTime_{op}$ is not directly observable, we approximate C_{op} using the end-to-end response latency.

To instantiate the unified cost model, we provide separate formulations for GPU and CPU executions, considering the architectural and memory hierarchy differences between the two devices.

For GPU-based execution, the total cost C_{GPU} accounts for both the computation time $ExecTime_{GPU}$ and the data transfer overhead $TransCost_{GPU}$ incurred during model invocation:

$$ExecTime_{GPU} = \frac{ModelFLOPS}{FLOPS} \times nrows \tag{6}$$

$$TransCost_{GPU} = \frac{ModelSize}{MemBW} + \frac{ModelSize}{GPUBW} + \text{Latency} \tag{7}$$

where $ModelFLOPS$ denotes the total floating-point operations required by the model, $FLOPS$ reflects the peak computational throughput of the GPU, $nrows$ is the number of input samples processed or inferred in the batch, $ModelSize$ refers to the size of the model in bytes, $MemBW$ represents the bandwidth of main memory, $GPUBW$ measures the transfer speed from main memory to GPU memory, and $Latency$ encompasses I/O or network latency.

In contrast, CPU execution avoids explicit memory hierarchy transitions, leading to a simplified cost expression:

$$ExecTime_{CPU} = \frac{ModelFLOPS}{FLOPS} \times nrows \tag{8}$$

$$TransCost_{CPU} = \frac{ModelSize}{MemBW} \tag{9}$$

The key distinction is that CPU execution avoids memory–GPU transfer, making it more efficient for small tasks, while GPUs offer higher throughput for compute-intensive workloads.

Device Selection Strategy: To determine the optimal execution device, we adopt a cost-based decision framework derived from the unified performance model:

$$\text{Device} = \begin{cases} \text{GPU} & \text{if } C_{GPU} < C_{CPU} \\ \text{CPU} & \text{otherwise} \end{cases} \tag{10}$$

This criterion ensures that the execution platform is selected based on a quantitative assessment of end-to-end cost, incorporating both computation time and data transfer time. By leveraging this analytical model, the system can dynamically allocate operators to the most suitable device, adapting to variations in model size, data volume, and system bandwidth.

Pipeline Processing: To further improve throughput during inference tasks, MorphingDB employs advanced pipeline techniques to streamline the execution of complex inference workflows. One critical challenge is identifying dependency relations between various operators, ensuring that tasks are processed in an optimal order without introducing bottlenecks in the DAG. We propose a dependency analysis algorithm based on the DAG to optimize the execution flow. The details are depicted in Algorithm 1. The algorithm first constructs a dependency map of operators by analyzing the edges between nodes (lines 3-5) and classifying dependency relationships between operators (lines 6-12). It then schedules the operators for execution by performing a topological sort using a Depth First Search (DFS) approach (lines 13–15). This method allows the system to prioritize operators with higher computational demands or critical operators, facilitating effective task scheduling and parallelization.

Batch Inferences: The inference function is reformulated as a window function to improve the throughput, with the parameter *batch_size*. The introduction of *batch_size* allows for flexibility in the inference process. When set to 1, the operator function is similar to a non-batch inference operator that processes one data point at a time.

To support the batch inferences in PostgreSQL, we modify the kernel's window function in three aspects: (1) **Window Data Aggregation**: Raw data from underlying PostgreSQL operators is duplicated into an intermediate state to extend its lifecycle. This ensures data is retained long enough for batch processing to improve efficiency during inferences. (2) **Batch Inference Execution**: Once

Algorithm 1 The Algorithm of Pipeline Dependency Discovery

Input: DAG $G = (V, E)$ where V is the set of operators, and E represents dependencies
Output: Execution order σ
for all $v \in V$ **do**
  $\mathcal{D}(v) \leftarrow \{u \in V \mid (u, v) \in E\}$
end for
for all $(u, v) \in E$ **do**
  if is_data_dependency(u, v) **then**
    Label (u, v) as "Data Dependency"
  else
    Label (u, v) as "Control Dependency"
  end if
end for
for all $v \in V$ **do**
  $\sigma \leftarrow$ DFS(v)
end for
return σ

a batch is filled, the system triggers the inference process. The raw data is processed in parallel using a thread pool to convert into input tensors. These tensors are combined along the batch dimension, maximizing throughput and resource utilization by leveraging hardware optimizations in model inference. (3) **Data Cleanup and Result Caching**: After completing inference, the results are transformed back into a PostgreSQL-compatible format, with tensors and other data types cached in an intermediate state for continued processing. Previous raw data is efficiently released in this phase, and parallelization ensures that this cleanup step sustains high throughput across batch operations.

Designing an optimal batch size for inferences is crucial to maximizing throughput and maintaining efficient resource utilization. Smaller batch sizes offer higher concurrency but lead to suboptimal use of computational resources, while larger batch sizes improve throughput but reduce concurrency due to higher memory. To address this challenge, MorphingDB designs a cost model that considers several factors to automatically select an optimal batch size. The model will balance the trade-off between throughput (T) and latency (L) while accounting for resource utilization (R), such as memory, computational power (e.g., CPU or GPU), and concurrency (C). Given a batch size B, the cost $C(B)$ can be calculated as:

$$C(B) = \arg\max_{B}(F(L(B), R(B), C(B), T(B))) \tag{11}$$

To delve deeper into the implications of different batch size configurations, we conduct experimental studies to analyze the impact of varying batch sizes, as detailed in Section 6.6. Our experimental findings reveal that batch size settings ranging from 8 to 32 typically yield optimal inference efficiency.

6 Evaluation

6.1 Experiment Setup

In this section, we detail the datasets, methodologies, and hardware configurations for evaluating the performance of our system. Moreover, the MorphingDB codebase is publicly available at: https://github.com/MorphingDB/MorphingDB.

Table 2: Nine datasets in three domains

Type	Dataset	#Dimension	Dataset Size
Series	YearPredict[18]	90 cols	515,345 rows
	Slice[24]	384 cols	53,500 rows
	Swarm[4]	2400 cols	24,017 rows
Image	ImageNet[36]	3*224*224	14,197,122 images
	CIFAR-10[35]	3*224*224	60,000 images
	Stanford Dogs[33]	3*224*224	20,580 images
NLP	SST-2[47]	4*128	70,042 sentences
	IMDb[40]	4*128	50,000 sentences
	Financial[41]	2*133	4,840 sentences

Datasets: We employ nine widely-used open datasets to benchmark the performance of AI-powered databases in Table 2. These datasets encompass three distinct types of inference tasks: Series tasks, Image tasks, and NLP tasks. The YearPredict dataset [18] includes audio features paired with song release years from 1922 to 2011. The Slice dataset [24] contains features from CT scans with a numeric class indicating slice position within the body. The Swarm dataset [4] captures high-dimensional features of animal swarm behavior, classified into binary motion categories. We also perform image inference tasks such as image classification on ImageNet[36], CIFAR-10[35], Stanford Dogs[33] datasets. SST-2[47], IMDb[40], Financial[41] are popular datasets for NLP inferences, i.e., binary sentiment classification tasks.

Hardware configuration: We utilize an Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz with 16 threads and 32GB of memory as the CPU server to evaluate efficiency and performance using the CPU. Additionally, we assess the efficiency and performance using an NVIDIA RTX 3090 GPU with 24GB of memory, on an Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz with 40 threads and 32GB of memory as the GPU server.

Baselines: We compare several open-source AI-native databases that integrate machine learning capabilities to support various algorithms. For example, functional integration system EvaDB [8], kernel-level integration systems Madlib [7] and GaussML [25] are considered for efficient model prediction. We also consider whether the platforms are open-source, reproducible, and structurally comparable to MorphingDB. For instance, MindsDB is excluded since MindsDB and EvaDB are similar in that both serve as middleware frameworks with ML frameworks. In addition, we evaluate our model selection strategy against widely adopted open-source AutoML platforms. AutoKeras [30] leverages Neural Architecture Search to automatically identify optimal deep learning models and hyperparameters. AutoSklearn [22] performs the selection and tuning of machine learning algorithms for a variety of tasks, while AutoGluon [50] processes multiple data modalities with GPU acceleration to handle more complex tasks.

Workloads: We define flexible, general-purpose templates (detailed in our Supplement File) within the database to support various AI inference workloads, including series classification and regression, NLP sentiment classification, and image classification tasks. For each dataset, we construct a representative set of tasks aligned with the dataset's characteristics and systematically generate 100+

queries using predefined label configurations. For time-series regression tasks, the query workload is relatively lightweight, primarily comprising selection, projection, window functions, and prediction. In comparison, NLP sentiment analysis and image classification introduce higher computational complexity by incorporating additional operations such as aggregation, group-by, and inference operators, which increase processing overhead and intermediate state management. Furthermore, multi-modal inference tasks further extend this complexity by performing cross-modal data fusion through join operations across heterogeneous inputs (e.g., text and image), followed by window functions applied to the integrated feature space. We run each query at least three times and take the average to ensure the robustness and consistency of the evaluation. However, when we observe higher variance (e.g., >20%)—often due to dynamic resource contention—we perform additional trials to ensure statistical stability and report representative averages.

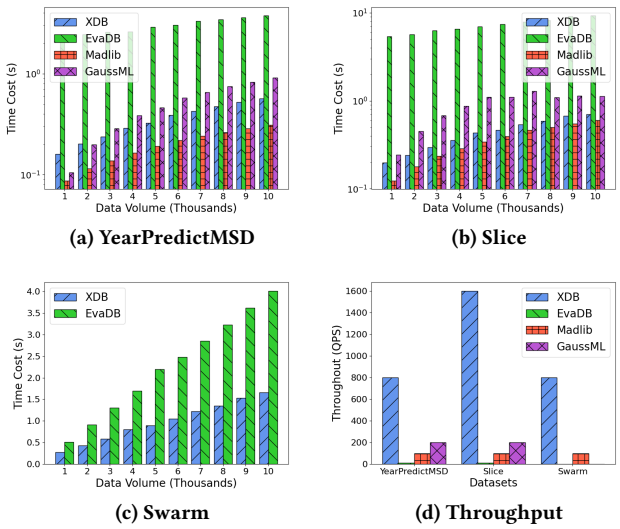
Structure of Experimental Trials: To comprehensively evaluate MorphingDB's performance across a broad range of scenarios, particularly to validate its robustness and efficiency under diverse task modalities, including series, NLP, and image inference workloads. We conduct multiple groups of experiments: (1) **Overall performance evaluation** assesses the integrated behavior of MorphingDB under nine datasets across three domains: series, NLP, and image, to validate the effectiveness and robustness in different scenarios. (2) **Component-level analysis** focuses on core modules such as model storage, model selection, and CPU-GPU Placement, requiring controlled trials to isolate the effect of each mechanism. (3) For **ablation studies**, we select one representative dataset per domain (e.g., Slice for time series, SST-2 for NLP, CIFAR-10 for image) to control the complexity and space constraints.

6.2 Efficiency and performance Study

In this section, we conduct a comprehensive efficiency and performance evaluation of MorphingDB across three domains using nine benchmark datasets. The results are illustrated as follows.

6.2.1 Series Tasks. We compare the performance with EvaDB, GaussML, and Madlib on series inference tasks across three datasets with different data volumes. Figure 6a and 6b illustrate the performance of **series classification tasks** on *YearPredic* dataset (90 columns) and *Slice* dataset (380 columns), respectively. As data volume increases, inference time rises across all systems in both cases. EvaDB demonstrates the highest inference time for both datasets, indicating less efficient handling of classification tasks on low-dimensional data. Madlib and MorphingDB, however, show close efficiency, suggesting both systems are well-suited to moderate-dimensional data, while GaussML lags slightly due to its less optimized query execution and data access methods. We also study the performance of **regression tasks** for the **high-dimensional** Swarm dataset (2400 columns) shown in Figure 6c. Notably, both EvaDB and GaussML encountered issues with PostgreSQL's column limit on the Swarm dataset, resulting in execution errors.

Figure 6d analyzes the throughput of MorphingDB and other systems, and we can find that MorphingDB achieves at least a **4x higher** throughput compared to other systems due to its implementation of batch inference within a pipelined architecture. MorphingDB achieves the highest throughput, even though its



(a) YearPredictMSD (b) Slice

(c) Swarm (d) Throughput

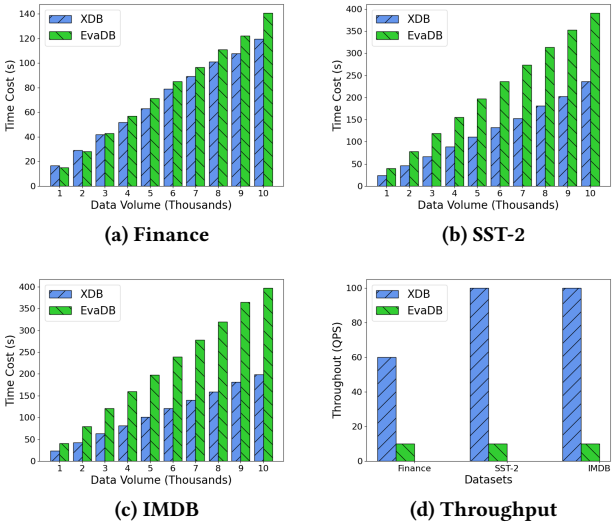
Figure 6: Performance comparison on Series Tasks

per-query latency is slightly higher than Madlib. This is primarily because MorphingDB employs optimized batch inference pipelines (e.g., DAG-based operator scheduling and vector sharing), which allow it to amortize overhead across multiple concurrent queries and maximize hardware utilization. GaussML, despite its longer average latency, GaussML's throughput improvements primarily benefit from internal optimizations such as native operator fusion and are driven by hardware-level acceleration for specific workloads. By processing batches of data in parallel, MorphingDB minimizes the idle time between stages, maximizing hardware utilization and reducing the overall latency per query.

6.2.2 NLP Tasks. We evaluate the inference efficiency of natural language processing tasks, specifically focusing on **sentiment classification** using the *ALBERT* model. Due to the lack of support for this model in GaussML and Madlib, EvaDB is selected as our baseline for comparison. The experiments are conducted with a batch size of 16 to ensure consistent evaluation conditions for both systems. As shown in Figure 7, MorphingDB demonstrates a significant improvement in inference efficiency across the three datasets, achieving a relative enhancement ranging from 10% to 50% compared with EvaDB, largely because of the vector sharing and batch pipeline optimization.

Notably, in the *Finance* dataset in Figure 7a, the performance advantage of MorphingDB is less pronounced. This can be attributed to the Finance dataset consists primarily of shorter sentences, reducing computational complexity and minimizing the processing time difference between MorphingDB and EvaDB. However, MorphingDB still has a higher throughput than EvaDB shown in Figure 7d. These findings indicate MorphingDB's capability to handle NLP tasks with particularly strong performance on longer, more complex text inputs.

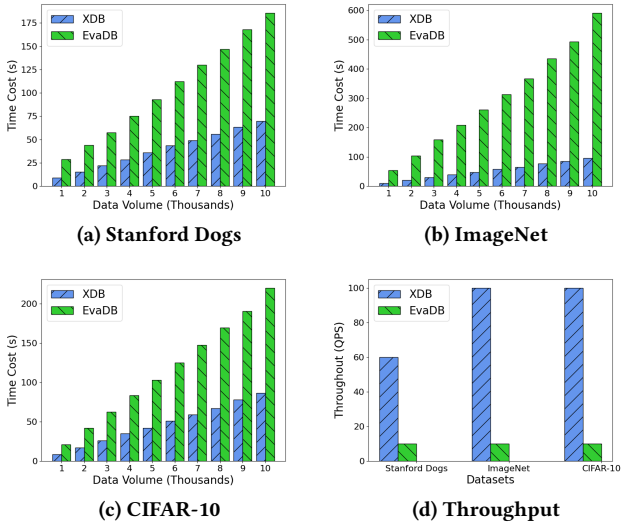
6.2.3 Image Tasks. We further evaluate the **image classification** efficiency of MorphingDB using the AlexNet, ResNet-18, and GoogleNet



(a) Finance (b) SST-2

(c) IMDB (d) Throughput

Figure 7: Performance comparison on NLP Tasks



(a) Stanford Dogs (b) ImageNet

(c) CIFAR-10 (d) Throughput

Figure 8: The performance comparison on Image datasets

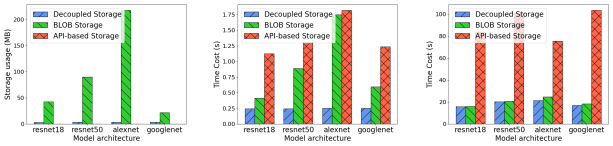
models across three benchmark datasets: Stanford Dogs, ImageNet, and CIFAR-10. Since GaussML and Madlib lack support for image classification tasks, we only compare MorphingDB and EvaDB. As shown in Figure 8, in comparison with EvaDB, MorphingDB achieves a substantial reduction in inference time, with an average decrease exceeding 70% across the three datasets. This performance advantage is primarily due to MorphingDB's storage of pre-embedding feature vectors directly within the database, contrasting with EvaDB's reliance on raw binary image data stored outside a database system. By leveraging PostgreSQL's optimized data management, MorphingDB significantly reduces retrieval and access latency, which contributes to faster data handling and processing.

For instance, on the *Stanford Dogs* dataset, MorphingDB achieves a substantial efficiency gain over EvaDB. This improvement is attributed to MorphingDB's vectorized storage format, which not only enables efficient organization and retrieval of image embeddings but also supports optimized inference on high-dimensional data. In contrast, EvaDB relies on image data storage, which incurs additional retrieval and decoding overhead, which heavily impacts performance on larger image datasets.

6.3 Effect of Model Storage

We conduct a comparative analysis of the model storage strategies: In-Database models, and API-based/external models. Specifically, we assess their storage usage, model loading overhead, and inference time associated with each approach.

Storage Usage. Figure 9a illustrates the model storage usage for different storage types. We find that BLOB-based storage results in the highest disk usage, as it stores the entire model—including both architecture and parameters. Decoupled storage, by contrast, only records the updated layers while reusing the base model, thus significantly reducing redundancy and storage cost. In comparison, API-based methods have negligible storage overhead, as the



(a) Storage Usage (b) Loading time (c) Inference time

Figure 9: Comparison of model storage strategies

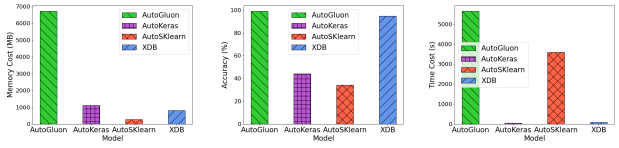
model is deployed externally and only minimal metadata (e.g., URL, version) needs to be stored locally.

Model loading time. Figure 9b compares the model loading time without inference across different storage strategies. BLOB-based models have a higher loading time, as the entire model binary must be deserialized and reconstructed at once. Decoupled storage mitigates this bottleneck by separating the architecture and parameter files, allowing partial loading and even parallel access to different model components. This significantly reduces initialization latency, especially in dynamic or resource-constrained environments. For the API-based method, it mainly involves lightweight service preparation, such as metadata retrieval and connectivity checks.

Model inference time. Figure 9c presents the inference performance when processing 10,000 records on a GPU. API-based models incur the highest latency, primarily due to the overhead of HTTP-based communication with external services. Both BLOB-based and decoupled in-database models operate locally, resulting in significantly lower inference times.

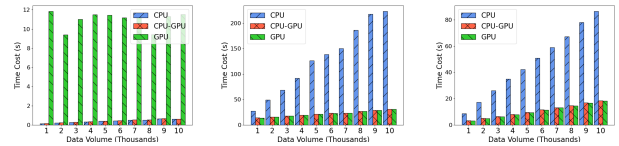
6.4 Effect of Model Selection

We compare our model selection algorithm of MorphingDB against AutoGluon, AutoKeras, and AutoSklearn in terms of resource consumption, processing time, and accuracy. Figure 10 presents a



(a) Memory Cost (b) Accuracy (c) Time Cost

Figure 10: Comparison of AutoML platforms



(a) Slice (b) SST-2 (c) CIFAR-10

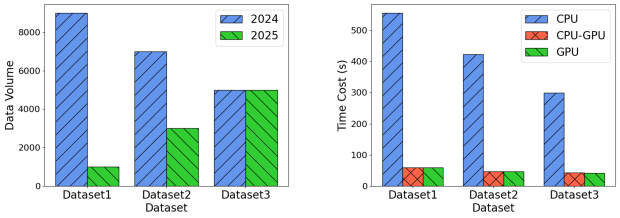
Figure 11: Comparison of different devices

comparative evaluation based on the CIFAR-10 dataset. *AutoGluon* achieves the highest accuracy among the evaluated platforms, demonstrating its capability to identify optimal model architectures and hyperparameters through the two-stage selection. However, this accuracy is attained at the cost of substantial computational resources—both in terms of memory usage and extended training durations. Consequently, this method is highly effective in resource-rich environments where accuracy is paramount. *AutoSklearn* offers a resource-efficient alternative, consuming significantly less memory compared to its counterparts. Despite this advantage, its accuracy is comparatively lower, which can restrict its utility in applications requiring high precision. Moreover, the training time remains relatively high, despite its lower memory usage. *AutoKeras* provides a balanced approach by delivering rapid training times alongside moderate memory usage. This makes it particularly attractive for applications where speed is critical. Nonetheless, its accuracy does not match that of AutoGluon or MorphingDB, suggesting that AutoKeras is more suitable for preliminary analyses or tasks where immediate results are prioritized over achieving the highest possible accuracy. MorphingDB demonstrates a robust balance among accuracy, resource consumption, and time cost. By leveraging transfer learning, MorphingDB incorporates pre-trained models that are subsequently fine-tuned for specific tasks. This strategy significantly reduces processing time and memory usage compared to models trained from scratch, while still achieving high accuracy and making it particularly well-suited for environments where both performance and resource constraints are critical.

6.5 Effect of CPU-GPU Placement

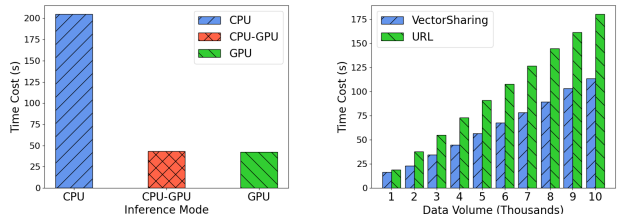
We evaluate the effectiveness of our cost-based CPU-GPU placement strategy across various dimensions: (1) heterogeneous task types with different operators, cardinality, and data volumes; (2) varying data skew and complex multi-modal queries.

Heterogeneous task types. We first evaluate inference performance across diverse task types—series, text, and image data—which represent varying model complexities, cardinality, and different operators. Figure 11 shows that CPU outperforms GPU in series tasks



(a) Data Distribution (b) Time Cost

Figure 12: Comparison of data skew.



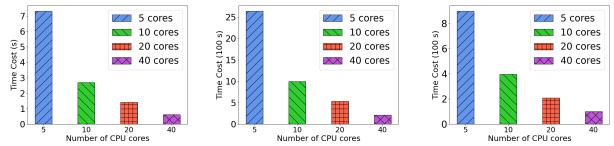
(a) Multi-modal (b) Vector Sharing

Figure 13: Multi-modal and Vector Sharing

due to the minimal computational load and the high overhead introduced by memory-to-GPU data transfer. The cost of transferring data to the GPU outweighs the potential gains in computation speed. Text and image tasks illustrated in Figures 11b and 11c benefit from GPU acceleration. These tasks involve more computationally intensive models, where the GPU's parallelism and higher throughput compensate for the initial data transfer overhead. Our cost model accurately identifies the lower-latency execution device, achieving near-optimal performance with minimal deviation.

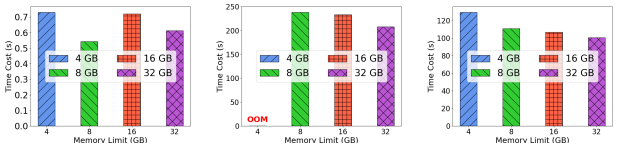
Data skew. We investigate the effectiveness of the cost model under varying degrees of data skew, where input data is unevenly distributed across temporal partitions. As illustrated in Figure 12a, we construct three datasets in which records from the year 2024 account for 90%, 70%, and 50% of the data, respectively. SQL queries are issued to filter and perform inference exclusively on 2024 data. Figure 12b compares inference performance across CPU, GPU, and CPU-GPU co-processing. The results demonstrate that even under skewed distributions, the cost model consistently selects the optimal device, effectively adapting to data locality and operator selectivity to minimize inference latency.

Multi-modal. We leverage selection and join operators to enable cross-modal data fusion by integrating image recognition and text reasoning models along with window functions applied to the data inputs. Figure 13a shows strong adaptability by assigning heterogeneous devices (e.g., GPU for image models and CPU for lightweight text models) to different sub-tasks. This device-aware execution enables global latency minimization, validating the effectiveness of our model in complex, heterogeneous inference scenarios.



(a) Slice (b) SST-2 (c) CIFAR-10

Figure 14: Comparison of CPU configs.



(a) Slice (b) SST-2 (c) CIFAR-10

Figure 15: Comparison of memory configs.

6.6 Ablation study

In this section, we conduct experiments to validate the advantage of pre-processing vector-sharing storage, pipeline-based batch inference, and resource allocation configurations on MorphingDB performance.

6.6.1 Effect of Vector Sharing. This experiment is designed to assess inference efficiency by comparing two approaches: sharing vector data as pre-processed and embedding vectors within the database versus referencing images by URLs. As shown in Figure 13b, sharing vectors directly in the database substantially improves efficiency. This advantage stems from the reduced latency associated with in-database vector access, as it eliminates the overhead of external data retrieval and minimizes the delays linked to URL-based storage. By centralizing data within the database, vector storage enhances both retrieval speed and overall inference throughput, making it a superior approach for AI-driven tasks.

6.6.2 Effect of Batch Size. We also investigate the performance of various batch sizes for our optimization strategy - batch inference in pipeline. In our experiments shown in Table 3, we evaluate MorphingDB's inference performance (time cost) across different batch sizes, including 4, 8, 16, 32, 64, and 128. The results reveal that there is an optimal batch size range that maximizes computational efficiency and high utilization, beyond which performance gains diminish or even degrade due to resource constraints. We can find that batch size 8 yields the best performance for the *ImageNet* dataset, while batch size 16 is optimal for the *Finance* dataset. Choosing the most appropriate batch size is essential to balancing concurrency and resource usage. Smaller batch sizes provide higher concurrency, enabling rapid response times for individual queries but potentially underutilizing computational resources. Conversely, larger batch sizes yield higher throughput by fully utilizing the hardware, but they may limit concurrency.

6.6.3 Effect of Different resource allocations. We evaluate task performance by imposing different resource allocation limits. On the GPU server, we restrict the CPU and memory usage of the MorphingDB Docker image. The experiments are conducted using the same model and dataset across three tasks: the slice dataset for the series task, the Finance dataset for the text task, and the CIFAR-10 dataset for the image task. As shown in Figure 14, after limiting the number of CPUs used to 5, 10, 20, and 40, the model processes 10,000 rows of data across different tasks, and the inference time is significantly reduced, with the reduction being non-linear. It is worth noting that when limiting memory usage to 4GB, 8GB, 16GB, and 32GB, the high memory demand for the text and image tasks led to out-of-memory (OOM) errors. Specifically, the text task encounters OOM under 4GB memory conditions. Meanwhile, for image and text tasks, the inference time decreases with increasing memory, and for series tasks, the inference time remains roughly the same in Figure 15.

Table 3: Effect of batch size

Dataset	Batch Size (Inference Time in Seconds)					
	4	8	16	32	64	128
ImageNet	108.31	**81.34**	86.72	84.49	91.63	106.80
Finance	180.35	136.86	**119.08**	135.06	143.40	192.12

7 Related Works

7.1 In-database AI Analytics

In-database AI analytics represents a paradigm shift where artificial intelligence (AI) and machine learning (ML) capabilities are directly integrated within the database system, enabling seamless data analysis and model inference without external computation. This fusion can be approached through two primary avenues: **Functional Integration** and **Kernel-Level Integration**.

The first one is **Functional Integration**, focusing on integrating external AI/ML frameworks, such as TensorFlow or PyTorch, or utilizing user-defined functions (UDFs) to execute machine learning tasks in database systems [2, 3, 5, 6, 8, 34, 42, 58]. Users can leverage SQL queries, UDFs, or APIs to invoke these models and perform predictive analytics directly within the database, such as MindsDB [3], EvaDB [8], and Raven [31, 42]. MindsDB [3] is an open-source platform that integrates ML capabilities directly into databases, allowing users to train, test, and deploy models without leaving the database environment. EvaDB [8] offers a SQL-based interface for building AI applications over both structured and unstructured data, supporting tasks such as regression, classification, image recognition, and generative AI. Klabe et al. [34] explore integrating ML inference into databases via user-defined functions (UDFs) and external ML runtime APIs, enhancing flexibility and interoperability with existing AI tools. Raven [31, 42] is a production-ready system that co-locates data and ML runtimes, using a unified graph abstraction for cross-runtime optimization and efficient query execution across heterogeneous hardware.

Kernel-level integration, on the other hand, by incorporating AI and ML algorithms into the core of the database engine, optimizing query processing and computational efficiency [1, 7, 9, 10,

25, 29, 39, 44, 44, 52, 58]. Madlib [7] facilitates advanced analytics and ML capabilities within a DBMS, enhancing performance and streamlining the analytics workflow. By leveraging distributed computing frameworks like Apache Hadoop and Apache Spark, Madlib enables efficient processing of large datasets without the need to transfer data between systems. MLog [39] enables declarative model training and prediction via SQL-like syntax, simplifying the specification and execution of ML workflows within a DBMS. NeurDB [58] integrates AI operators—such as training, inference, and fine-tuning—directly into the DBMS, supporting adaptive analytics under data and workload drift. InferDB [44] optimizes in-database inference by transforming ML models into index-like structures, enabling low-overhead, index-based predictions. GaussML [25] offers a full-stack ML pipeline within the DBMS, covering preprocessing, training, evaluation, and deployment to reduce data movement and improve system efficiency.

MorphingDB adopts the second integration approach by embedding inference algorithms as native query operators, enabling seamless execution of model inference within the query pipeline. It supports hardware acceleration (e.g., GPUs) and extends the engine with AI-native data structures—such as tensors and vector formats—to ensure flexible model storage and efficient large-scale inference.

7.2 Automated Machine Learning

Automated Machine Learning (AutoML) [32] has emerged as a transformative approach in the field of artificial intelligence, aiming to streamline and automate the end-to-end process of applying machine learning to real-world problems without requiring deep expertise in data science or machine learning. The core task of AutoML is model selection, which is to train and evaluate several models and automatically identify the most suitable model for a given dataset and task without requiring manual intervention.

AutoSklearn [22] partitions the dataset into training and validation sets using the holdout method to estimate validation loss. It employs a bandit-based strategy, specifically successive halving (SH), to efficiently allocate resources across candidate pipelines. AutoKeras [30] leverages the evaluation information to define a search space that includes both neural architecture patterns and common hyperparameters. It also tunes hyperparameters from preprocessing steps and the training process to optimize model performance. AutoGluon [50] implements a two-stage selection procedure. It first selects the top five models for image, text, and tabular data based on their unimodal benchmarking performance, and subsequently performs a random search over combinations of these models along with additional hyperparameters. The optimal combination identified through this process serves as the default configuration. TRAILS [55] proposes an SLO-aware and resource-efficient two-phase model selection algorithm with a novel coordinator to leverage the strengths of both efficient training-free and effective training-based model selection.

In contemporary AutoML frameworks, model selection focuses on the evaluation metrics of neural architectures and hyperparameter selection [28], followed by full training on the target task which is computationally intensive. In contrast, our model selection approach aligns more closely with transfer learning [59], where the most suitable pre-trained model is selected for the target task. Compared to training from scratch, this paradigm not only leverages the rich knowledge learned from the source task but also significantly accelerates the training process.

8 Conclusion

In this paper, we present MorphingDB, a task-centric AI-native DBMS implemented as a PostgreSQL extension with LibTorch, designed to automate model management, selection, and inference within the database engine. MorphingDB introduces specialized schemas and data types with an Mvec representation for 3D tensors, supporting both BLOB-based all-in-one and decoupled architecture–parameter formats. Model selection in MorphingDB is formulated as a transfer learning problem, addressed through a two-phase framework. An offline embedding phase constructs a transferability subspace from historical tasks, and an online projection phase maps new tasks into this subspace to identify suitable models with minimal overhead.. Furthermore, two optimization strategies: pre-embedding with vectoring sharing and DAG-based batch pipelines, are proposed to improve the inference throughput. Extensive evaluation on nine public datasets spanning time-series forecasting, natural language processing, and image recognition demonstrates that MorphingDB achieves superior performance compared to existing AI-native DBMSs (e.g., EvaDB, MADlib, GaussML) and state-of-the-art AutoML frameworks (e.g., AutoGluon, AutoKeras, AutoSklearn), offering a robust trade-off between prediction accuracy, resource efficiency, and model selection latency.

Acknowledgments

This work was supported by the Key Research Program of Zhejiang Province (No. 2023C01037). It was also supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62572422.

References

[1] 2018. *Microsoft Azure SQL Database.* https://azure.microsoft.com/en-us/products/azure-sql/database/
[2] 2020. *Google Cloud AI Platform.* https://cloud.google.com/ai-platform
[3] 2020. *MindsDB.* https://mindsdb.com/
[4] 2020. Swarm Behaviour. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5N02J.
[5] 2021. *Amazon SageMaker.* https://aws.amazon.com/sagemaker/
[6] 2022. *IBM Db2 AI for z/OS.* https://www.ibm.com/products/db2-ai-for-zos
[7] 2023. *Apache MADlib.* https://madlib.apache.org
[8] 2023. *EvaDB.* https://evadb.ai/
[9] 2023. *Oracle Autonomous Database.* https://www.oracle.com/database/technologies/autonomous-database.html
[10] 2023. *SAP HANA.* https://www.sap.com/products/hana.html
[11] 2024. *LibTorch.* https://pytorch.org/cppdocs/
[12] 2024. *ONNX-Open Neural Network Exchange.* https://onnx.ai/
[13] 2024. *PostgreSQL.* https://postgresql.org/
[14] 2025. . https://chatgpt.com/
[15] 2025. *Hugging Face Hub.* https://huggingface.co
[16] 2025. *Ollama.* https://ollama.com/
[17] 2025. *Oracle AI Vector Search User's Guide.* https://docs.oracle.com/en/database/oracle/oracle-database/23/vecse/
[18] Thierry Bertin-Mahieux. 2011. Year Prediction MSD. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C50K61.
[19] Daniel Bolya, Rohit Mittapalli, and Judy Hoffman. 2021. Scalable diverse model selection for accessible transfer learning. *NIPS* 34 (2021), 19301–19312.
[20] Vincenzo Di Cicco, Donatella Firmani, Nick Koudas, Paolo Merialdo, and Divesh Srivastava. 2019. Interpreting deep learning models for entity resolution: an experience report using LIME. In *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management.* 1–4.

[21] Vijay Ekambaram, Arindam Jati, Nam Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. 2023. Tsmixer: Lightweight mlp-mixer model for multivariate time series forecasting. In *SIGKDD*. 459–469.

[22] M Feurer, K Eggensperger, S Falkner, J T Springenberg, A Klein, and F Hutter. 2020. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074* (2020).

[23] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. 2021. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430* (2021).

[24] Franz Graf, Hans-Peter Kriegel, Matthias Schubert, Sebastian Poelsterl, and Alexander Cavallaro. 2011. Relative location of CT slices on axial axis data set. *UCI Machine Learning Repository* (2011).

[25] Lijie Xu Shifu Li Jiang Wang Wen Nie Guoliang Li, Ji Sun. 2024. GaussML: An End-to-End In-database Machine Learning System. *ICDE* (2024).

[26] Fengxiang He, Tongliang Liu, and Dacheng Tao. 2020. Why resnet works? residuals generalize. *IEEE TNNLS* 31, 12 (2020), 5349–5362.

[27] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask R-CNN. In *ICCV*. 2961–2969.

[28] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems* 212 (2021), 106622.

[29] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Roehm, and Carsten Binnig. 2020. DB4ML-an in-memory database kernel with machine learning support. In *SIGMOD*. 159–173.

[30] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *SIGKDD*. ACM, 1946–1956.

[31] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. 2020. Extending Relational Query Processing with ML Inference. In *CIDR*.

[32] S K Karmaker, M M Hassan, M J Smith, et al. 2021. AutoML to Date and Beyond: Challenges and Opportunities. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–36.

[33] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. 2012. Novel Dataset for Fine-Grained Image Categorization : Stanford Dogs. https://api.semanticscholar.org/CorpusID:3181866

[34] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2023. Exploration of Approaches for In-Database ML. In *EDBT*. 311–323.

[35] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *NIPS* 25 (2012).

[37] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *ICLR*.

[38] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI meets database: AI4DB and DB4AI. In *SIGMOD*. 2859–2866.

[39] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *PVLDB* 10, 12 (2017), 1933–1936.

[40] Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *ACL*. 142–150.

[41] Pekka Malo, Ankur Sinha, Pekka Korhonen, Jyrki Wallenius, and Pyry Takala. 2014. Good debt or bad debt: Detecting semantic orientations in economic texts. *Journal of the Association for Information Science and Technology* 65, 4 (2014), 782–796.

[42] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end optimization of machine learning prediction queries. In *SIGMOD*. 587–601.

[43] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *MICCAI 2015*. Springer, 234–241.

[44] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *PVLDB* 17, 8 (2024), 1830–1842.

[45] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *ArXiv* abs/1910.01108 (2019).

[46] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *ICLR*.

[47] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *EMNLP*. Association for Computational Linguistics, Seattle, Washington, USA, 1631–1642. https://www.aclweb.org/anthology/D13-1170

[48] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: a serverless vector data management system using cloud functions. *SIGMOD* 2, 3 (2024), 1–26.

[49] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *CVPR*. 2818–2826.

[50] Z Tang, H Fang, S Zhou, et al. 2024. AutoGluon-Multimodal (AutoMM): Supercharging multimodal AutoML with foundation models. *PMLR* 256 (2024).

[51] Xin Wang, Hui Guo, Shu Hu, Ming-Ching Chang, and Siwei Lyu. 2023. Gan-generated faces detection: A survey and new perspectives. *ECAI 2023* (2023), 2533–2542.

[52] Yi Wang, Yang Yang, Weiguo Zhu, Yi Wu, Xu Yan, Yongfeng Liu, Yu Wang, Liang Xie, Ziyao Gao, Wenjing Zhu, et al. 2020. SQLFlow: A Bridge between SQL and Machine Learning. *CoRR* (2020).

[53] Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-pack: Packed resources for general chinese embeddings. In *SIGIR*. 641–649.

[54] Jiadong Xie, Jeffrey Xu Yu, and Yingfan Liu. 2025. Fast Approximate Similarity Join in Vector Databases. *SIGMOD* (2025).

[55] Naili Xing, Shaofeng Cai, Gang Chen, Zhaojing Luo, Beng Chin Ooi, and Jian Pei. 2024. Database native model selection: Harnessing deep neural networks in database systems. *PVLDB* 17, 5 (2024), 1020–1033.

[56] Runjie Yu, Weizhou Huang, Shuhan Bai, Jian Zhou, and Fei Wu. 2025. AquaPipe: A Quality-Aware Pipeline for Knowledge Retrieval and Large Language Models. *SIGMOD* 3, 1 (2025), 1–26.

[57] Kaiyu Yue, Bor-Chun Chen, Jonas Geiping, Hengduo Li, Tom Goldstein, and Ser-Nam Lim. 2024. Object Recognition as Next Token Prediction. In *CVPR*. IEEE, 16645–16656. https://doi.org/10.1109/CVPR52733.2024.01575

[58] Zhanhao Zhao, Shaofeng Cai, Haotian Gao, Hexiang Pan, Siqi Xiang, Naili Xing, Gang Chen, Beng Chin Ooi, and et al. 2025. NeurDB: On the Design and Implementation of an AI-powered Autonomous Database. In *CIDR*.

[59] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.

Received April 2025; revised July 2025; accepted August 2025