Database as Runtime

Compiling LLMs to SQL for In-database Model Serving

Sun, Wenbo; Li, Ziyu; Hai, Rihan

DOI
10.1145/3722212.3725093

Publication date
2025

Document Version
Final published version

Published in
SIGMOD-Companion 2025 - Companion of the 2025 International Conference on Management of Data

Database as Runtime: Compiling LLMs to SQL for In-database Model Serving

Wenbo Sun
w.sun-2@tudelft.nl
Delft University of Technology
the Netherlands

Ziyu Li
zli17nl@gmail.com
Delft University of Technology
the Netherlands

Rihan Hai
r.hai@tudelft.nl
Delft University of Technology
the Netherlands

Abstract

Deploying large language models (LLMs) often requires specialized hardware and complex frameworks, creating barriers for CPU-based environments with resource constraints. These systems, common in air-gapped or edge scenarios, lack support for maintenance due to security, budget, or technical limits. To address this, we introduce TranSQL$^+$, a compiler that translates LLM inference into SQL queries, enabling deployment on relational databases. By converting transformer operations into relational algebra, TranSQL$^+$ generates vector-oriented SQL queries that leverage native database features (buffer management, indexing) to manage computations without hardware accelerators or deep learning frameworks. Demonstrated with the LLaMA3.1 8B model on DuckDB, results show relational databases can effectively serve LLMs, reducing deployment barriers and expanding access to advanced AI.

CCS Concepts

• **Information systems** → **Structured Query Language**; *Query operators*; • **Computing methodologies** → *Natural language generation.*

Keywords

Large Language Models, Relational Database, Query Processing

ACM Reference Format:
Wenbo Sun, Ziyu Li, and Rihan Hai. 2025. Database as Runtime: Compiling LLMs to SQL for In-database Model Serving. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725093

1 Introduction

Large language models (LLMs) have shown remarkable effectiveness across various applications, including enterprise analytics [19], personalized edge computing [2]. However, many critical operational environments are constrained by CPU-based infrastructures, making GPU/TPU acceleration infeasible due to security, budget, or hardware limitations. Examples include air-gapped systems (e.g., healthcare and defense) [1], legacy enterprise servers, and cost-sensitive edge devices such as Raspberry Pi clusters. This diverse landscape of underpowered, CPU-only environments poses a significant challenge to deploying large-scale LLMs.

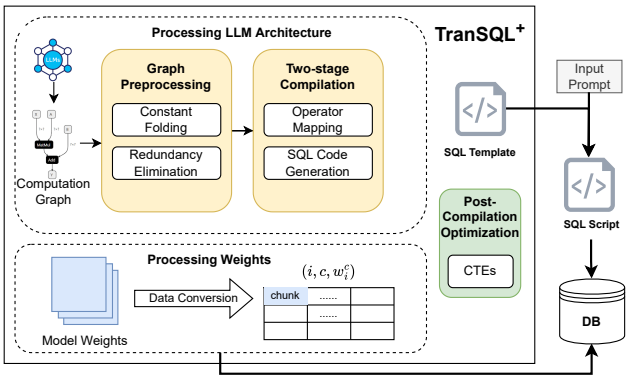

Figure 1: Workflow of the TranSQL$^+$ compiler, which consists of four major steps: data conversion, graph preprocessing, two-stage compilation and post-optimization.

Current model serving approaches typically use intermediate representation (IR)-based solutions [8, 14], which compile computations into portable IRs. While effective in stable hardware environments, these approaches require specialized backends and maintenance to accommodate OS/driver updates and instruction set evolution. For organizations with limited resources, the repeated engineering overheads become unsustainable, effectively barring them from accessing state-of-the-art AI capabilities.

Relational databases offer untapped potential for democratizing LLM deployment. They are widely deployed across computing environments—from lightweight SQLite instances[1] to enterprise-grade systems. With built-in capabilities for buffer management, indexing, and query optimization, they could theoretically manage LLM weights and computations without specialized hardware. Prior work, such as translating basic neural networks into SQL queries [9, 16], demonstrates early promise but lacks support for transformer-based architectures—specifically mechanisms like multi-head attention and dynamic key-value caching [12]. These omissions limit scalability and practical utility for modern LLMs.

Our previous research [18] has conceptually demonstrated that SQL is expressive enough to perform LLM inference entirely through SQL queries. In this paper, we propose a systematic solution, TranSQL$^+$, which supports translating LLM inference computation graphs into executable SQL queries. By operating on fine-grained operators (e.g., matrix multiplication and vector arithmetic) rather than monolithic layers, TranSQL$^+$ achieves **architecture-agnostic** compilation, supporting diverse neural network designs. Crucially, TranSQL$^+$ leverages databases' native memory management to automatically

[1] https://www.sqlite.org/

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725093

handle out-of-memory models via disk-backed relational tables, bypassing manual optimization efforts for resource-constrained environments.

We showcase our compiler using the LLaMA3.1 8B model [3], demonstrating end-to-end inference via DuckDB [13]. We show that databases can serve as portable runtimes for serving LLMs. This work not only reduces reliance on fragmented AI toolchains but also unlocks new possibilities for deploying advanced language models on existing infrastructure, democratizing access to cutting-edge AI.

2 Related Work

In-Database Machine Learning. Common in-database ML solutions often rely on user-defined functions (UDFs), as in PostgresML[2] and MADlib [4], where models run as opaque extensions. Our approach instead converts ML operations into standard SQL, eliminating external runtimes and enabling the database optimizer to see and plan each step. Earlier works [6, 15] also rewrite ML steps as queries but focus on simple linear models and require manual code rewrites. In contrast, we handle modern transformer-based architectures by mapping advanced operators like attention and feedforward layers into relational functions.

Deep Learning on Relational Platforms. Recent database research has investigated ways to represent deep learning operations via relational algebra. For instance, ModelJoin [7], SmartLite [10], and Dimitrije et al. [5] all embed matrix computations within relational systems but usually require substantial database-specific modifications, which can hinder portability across different frameworks.

Two notable developments are DuckBrain [16] and DL2SQL [9], which move toward tighter integration of deep learning with standard database engines. DuckBrain supports basic neural networks but lacks the capacity to handle large-scale architectures, serving primarily as a proof-of-concept. DL2SQL maps elementary CNN modules into SQL yet faces challenges with advanced operators like rotary embeddings or attention layers.

In contrast, our method focuses on a low-level operator mapping approach built around elementary arithmetic operations—such as matrix multiplications and elementwise transforms—allowing the compiler to handle a broad array of deep learning structures, including large language models. By leveraging the topological order of computational graphs, it systematically maps operators into SQL and reconstructs model structures for deployment within relational databases.

3 Compiling LLMs to SQL

To store model weights into relational databases, we represent matrices in a *chunked* format, adapting prior work on database-compatible linear algebra [5, 11]. To align with vector operation support in databases [13, 17], matrices $\mathbf{W} \in \mathbb{R}^{m \times n}$ are partitioned into fixed-size row chunks rather than tiles. Each row $\mathbf{w}_i$ is split into $\lfloor \frac{n}{chunk\_size} \rfloor$ chunks $\mathbf{w}_i^{(c)}$ and stored as database tuples:

$$(i,\ c,\ \mathbf{w}_i^{(c)})$$

[2]https://github.com/postgresml/postgresml

where $0 \le i < m$ and $0 \le c < \lfloor \frac{n}{chunk\_size} \rfloor$, and *chunk_size* is a hyperparameter.

Based on this representation, we propose a systematic method to compile the computational graph of LLMs for model serving into SQL queries, a format widely supported by relational databases. The process involves two stages: (1) *Operator Mapping*, where each computational graph node is converted into a high-level relational function, and (2) *SQL Code Generation*, where these relational functions are translated into executable queries in the target database's dialect.

3.1 Stage 1: Operator Mapping

In the first stage, operators are processed in topological order, extracting details such as operator type (e.g., `MatMul`, `Add`, `Reshape`), input/output shapes, and attributes like broadcast axes or chunk sizes. Each operator is mapped to a high-level relational function representing its equivalent relational primitives: π (projection), $\bowtie$ (join), or γ (group-by with aggregation). These relational functions provide an intermediate abstraction, reinterpreting neural operators in LLMs for execution in a relational environment.

We denote a *linear-algebraic function* over i operands as:

$$f\Big(\{O_1, \ldots, O_i\},\ \{\mathcal{F}_1, \ldots, \mathcal{F}_i\},\ \mathcal{S}\Big).$$

Each operand O_p (for $1 \le p \le i$) is associated with some *free dimensions* in a tuple $\mathcal{F}_p = (\ fd_1^p,\ fd_2^p,\ \ldots)$. $\mathcal{S} = \{\ sd_1,\ sd_2,\ \ldots,\ sd_n\}$ is the set of *shared dimensions* among these operands. Intuitively, the free dimensions remain as independent axes in the output, whereas the shared dimensions indicate indices to be matched or aggregated across the different operands.

We map f into a relational function

$$\mathcal{R}\Big(\{R_1, \ldots, R_i\},\ \{\text{keys}_1, \ldots, \text{keys}_i\},\ \text{keys}_{\text{join}}\Big)$$

where R_p is the chunk-based table representing operand O_p, keys_p is the relational counterpart of the free dimensions $\mathcal{F}_p$ (i.e., the row/column identifiers that remain in the final output), and $\text{keys}_{\text{join}}$ encodes the shared dimensions $\mathcal{S}$ as equi-join attributes for $\bowtie$. The resulting relational function is composed of standard algebraic operators (π, $\bowtie$, γ, and arithmetic).

By converting each neural operator into such relational function, the compiler forms a new graph whose edges point from input relations to output relations, and whose nodes are parameterized relational-algebraic operations. This SQL node graph directly reflects the original model structure while shifting the computational paradigm from matrix-based to relational-based primitives.

Matrix Multiplication Example. When the compiler encounters the operation $\mathbf{C} = \mathbf{AB}$ in a computational graph, suppose

$$\mathbf{A} \in \mathbb{R}^{m \times r}, \quad \mathbf{B} \in \mathbb{R}^{r \times n}, \quad \mathbf{C} \in \mathbb{R}^{m \times n}.$$

We identify the free dimensions fd_1^A and fd_1^B and the shared dimension sd_1 as follows: $fd_1^A = \big\{\, i \;\big|\; 0 \le i < m \big\}$, $fd_1^B = \big\{\, j \;\big|\; 0 \le j < n \big\}$, and $sd_1 = \big\{\, k \;\big|\; 0 \le k < r \big\}$.

Our system employs a *chunk-based* representation, and the dimension r is subdivided into chunks of size `chunk_size`. Concretely, the compiler replaces sd_1 with

$$C = \big\{ c \;\big|\; 0 \le c < \lfloor \tfrac{r}{\texttt{chunk\_size}} \rfloor \big\},$$

so that each chunk index c corresponds to a slice of size, `chunk_size`, within the original dimension r. The matrix multiplication can be

represented as follows:

$$Matmul(AB) \longmapsto \gamma_{(i,j),\ \mathrm{SUM}(\mathbf{a}^{(c)} \otimes \mathbf{b}^{(c)})} \left(R_A \ \bowtie_c \ R_B\right),$$

where R_A and R_B are the chunked relational tables for **A** and **B**. The join on c aligns matching row fragments, and the aggregation (γ) sums partial products to form each $\mathbf{c}_{ij}$. This example demonstrates how free dimensions (i, j) are retained in the output and the shared dimension k is "consumed" by the join and subsequent aggregation.

3.2 Stage 2: SQL Code Generation

SQL is a highly structured language. Once the required attributes and operands are identified, they can be combined with relational primitives to construct the final SQL query. Specifically, the compiler converts the *SQL node graph* into executable queries customized for the target database's SQL dialect, handling syntax variations and function compatibility across engines like PostgreSQL and DuckDB. Standard relational primitives (e.g., projections, joins, group-bys) are directly translated into `SELECT` clauses, while vector operations such as inner products or specialized aggregations are implemented using user-defined functions (UDFs) when necessary.

To optimize execution, the compiler can merge nodes into common table expressions or `CREATE VIEW` statements, reducing intermediate result overhead. For instance, element-wise operations can be fused into a single projection if the database supports efficient inline expressions. The final SQL script replicates the original LLM model's functionality, including complex operators like multi-head attention and reshaping, all within standard relational constructs.

3.3 Example: Self-Attention

In this section, we use the grouped query attention mechanism in LLama3 as an example to illustrate the translation of neural operators into SQL queries.

The Attention mechanism is expressed mathematically as

$$softmax(\frac{Q \cdot K^T}{\sqrt{d}})V,$$

comprising multiple matrix multiplications followed by a softmax normalization. To implement the matrix multiplications ($Q \cdot K^T$ and subsequent multiplication with V), we first join the query, key, and value embeddings tables:

```
SELECT token, head, row,
SUM(DOT(query_chunk, embedding)) AS q
FROM query[key,value]_weights AS A
JOIN embedding AS B ON A.col = B.col
GROUP BY row
```

Subsequently, the computed query and key vectors (Q, K) are combined and scaled by the square root of the head dimension. Since LLaMA3 employs grouped query attention, we group every four query vectors together when performing multiplication with key vectors:

```
SELECT Q.token, K.token, Q.head,
EXP(SUM(q * k) / sqrt(head_dim)) AS qk
FROM Q JOIN K ON Q.row = K.row AND Q.head//4 = K.head
GROUP BY Q.token, K.token, Q.head
```

The softmax operation is performed by normalizing each entry by the summation across tokens per head:

```
WITH summation AS (
  SELECT Q.head, Q.token, SUM(qk) AS s FROM QK
  GROUP BY Q.token, Q.head
)
SELECT Q.head, Q.token, K.token, qk/s
FROM QK JOIN summation ON Q.head = summation.head
AND Q.token = summation.token
```

4 TranSQL$^+$ Workflow

The compiler transforms LLM inference workflows into optimized SQL queries through four stages, as illustrated in Figure 1.

Data Conversion. Model weights are decomposed into relational tables using a *chunked* row-wise partitioning scheme. For a matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, each row $\mathbf{w}_i$ is split into C fixed-size chunks $\mathbf{w}_i^{(c)}$, stored as tuples $(i, c, \mathbf{w}_i^{(c)})$ to align with database vector operations. This ensures compatibility with systems lacking native matrix support while avoiding wide-table inefficiencies.

Graph Preprocessing. The compiler takes an ONNX computational graph as input. This graph undergoes two optimizations: 1) static evaluation of constant operations (e.g., bias additions, scalar multipliers) to embed them into weight chunks, which reduces runtime computations, and 2) elimination of redundant operators, such as reshape or expand, since relational projections inherently handle data layout in SQL.

Two-Stage Compilation. With the simplified computational graph, each ONNX operator is mapped to relational functions primitives, as introduced in Sec. 3.1. The relational functions are connected based on input and output as a SQL node graph, then converted to SQL queries, with creating necessary vector operation UDFs based on database's dialect.

Post-Compilation Optimization. To minimize execution overhead, Intermediate queries are fused into single statements using Common Table Expressions (CTEs), guided by the computational graph's topological order. This reduces temporary table creation and disk I/O.

5 Demonstration Scenarios

We illustrate TranSQL$^+$ using the `LLaMA3-8B` model and `DuckDB`, deployed on an AWS `c7i.2xlarge` instance. The demonstration is divided into three parts: *i)* displaying the inference computational graph along with its corresponding SQL node graph, which captures the relational functions generated during the compiler's initial phase; *ii)* showcasing SQL queries used to serve the model; and *iii)* accepting user-provided prompts to generate tokens.

5.1 Operator Mapping to Relational Functions

This demonstration begins by loading the `LLaMA3` model with user-selected scale/tile sizes from Block 1. During the compiler's initial phase, the model weights are partitioned into equal-sized vectors and stored in indexed Parquet files. These files are subsequently integrated into `DuckDB` as relational tables, thereby arranging the weights in a manner compatible with SQL operations.

In parallel, the model's ONNX-based computational graph is processed through operator mapping to relational functions. Participants can view: 1) the original computational graph displayed with `Netron`; 2) the relational functions mapped to neural operators; and 3) how these functions compose the `LLaMA3` modules, such as multi-head attention and rotary positional encoding.

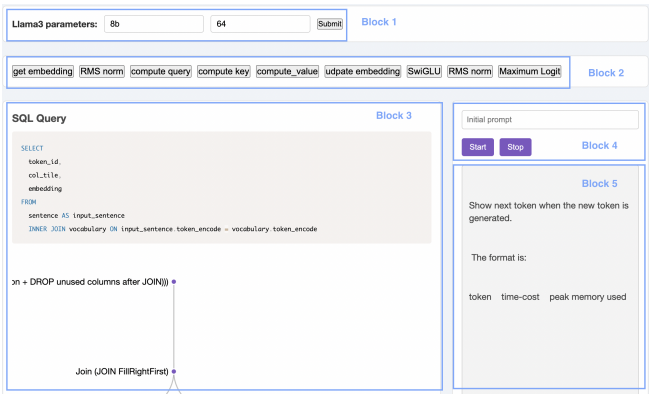

Figure 2: Screenshot of the demonstration interface. Participants specify the model scale and tile size and interact with LLMs through prompting.

5.2 Showcasing SQL Code for LLM Serving

Relational functions are translated into SQL statements. Through Block 2 of the interface (Figure 2), participants can: 1) inspect the query execution plans to see how the database manages LLM inference, and 2) choose specific `LLaMA3-8B` layers to examine the corresponding SQL queries and weights in the underlying tables.

5.3 Prompting and Text Generation

In the final phase, by entering any desired prompt in Block 4 and pressing the start button, the system will sequentially produce the next tokens. During token generation, the time usage and peak database memory consumption are displayed in Block 5, as shown in Figure 2. This mechanism demonstrates the resource footprint of an SQL-based LLM inference pipeline, highlighting how a relational database can handle memory and computation requirements effectively.

5.4 Preliminary Results

We evaluate an 8B model on a resource-limited hardware configuration (6 CPU cores, 8GB RAM), where the model size exceeds memory capacity. This setup allows us to assess the efficiency of the database's intrinsic buffer management. We benchmark against Llama.cpp[3]—a C++ framework supporting disk-memory hybrid weight loading, as illustrated in Figure 3.

6 Conclusion and Discussion

In this paper, we introduced TranSQL$^+$, a compiler that translates LLM inference computational graphs into queries for relational database systems. This method provides an alternative for deploying large-scale language models in environments with limited access to high-performance infrastructure or maintenance resources.

Using the LLaMA3.1 8B model, we demonstrated TranSQL$^+$'s ability to compile core neural operators, including matrix multiplication and multi-head attention, enabling end-to-end text generation within an RDBMS. The evaluation results highlight TranSQL$^+$'s potential in efficiently serving LLMs on resource-resource-constrainted hardware with minimal modifications.

[3]https://github.com/ggerganov/llama.cpp

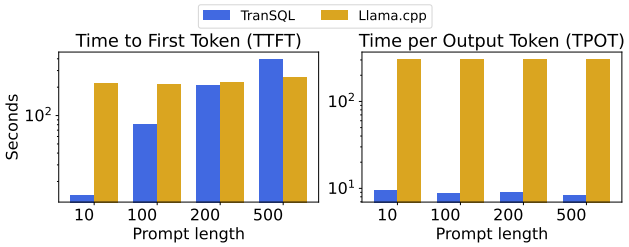

Figure 3: Response time of 8B model in varying prompt lengths. Our method shows 30x speedups over baseline regarding TPOT.

Acknowledgments

This publication was supported (in part) by Dutch Research Council (VI.Veni.222.439).

References

[1] Eugene Bagdasarian, Ren Yi, Sahra Ghalebikesabi, Peter Kairouz, Marco Gruteser, Sewoong Oh, Borja Balle, and Daniel Ramage. 2024. AirGapAgent: Protecting Privacy-Conscious Conversational Agents. In *CCS '24*. 3868–3882.
[2] Xin Luna Dong, Seungwhan Moon, Yifan Ethan Xu, Kshitiz Malik, and Zhou Yu. 2023. Towards Next-Generation Intelligent Assistants Leveraging LLM Techniques. In *SIGKDD '23*. 5792–5793.
[3] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and et al. 2024. The Llama 3 Herd of Models. *CoRR* abs/2407.21783 (2024). arXiv:2407.21783
[4] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1700–1711.
[5] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, et al. 2020. Declarative Recursive Computation on an RDBMS: or, Why You Should Use a Database For Distributed Machine Learning. *SIGMOD Rec.* 49, 1 (2020), 43–50.
[6] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-Database Learning Thunderstruck. In *DEEM'18* (Houston, TX, USA). Article 8, 10 pages.
[7] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2023. Exploration of Approaches for In-Database ML. In *EDBT'23*. 311–323.
[8] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, and et al. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization*. 2–14.
[9] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Feifei Li, and Gang Chen. 2022. A Comparative Study of in-Database Inference Approaches. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1794–1807.
[10] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. 2023. SmartLite: A DBMS-based Serving System for DNN Inference in Resource-constrained Environments. *Proc. VLDB Endow.* 17, 3 (2023), 278–291.
[11] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *SIGMOD '21*. ACM, 1222–1234.
[12] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *MLSys'23*.
[13] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD '19*. New York, NY, USA, 1981–1984.
[14] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: a new IR for machine learning frameworks. In *SIGPLAN '18*. ACM, 58–68.
[15] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD'16*. 3–18.
[16] Maximilian E. Schüle, Thomas Neumann, and Alfons Kemper. 2024. The Duck's Brain. *Datenbank-Spektrum* 24, 3 (2024), 209–221.
[17] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744. https://doi.org/10.14778/3685800.3685802
[18] Wenbo Sun, Ziyu Li, Vaishnav Srinidhi, and Rihan Hai. 2025. Database is All You Need: Serving LLMs with Relational Queries. In *EDBT 2025, Barcelona, Spain, March 25-28, 2025*. 1118–1121.
[19] Limiao Xie, Jianfeng Zhang, Yingying Li, Shan Wan, Xuequan Zhang, Mingjie Chen, and et al. 2024. Research and Application of Private Knowledge-based LLM in Standard Operating Procedure Scenarios of Enterprises. In *PRIS '24*. 82–86.