IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

AN INTELLIGENT ENTERPRISE DATA INTERFACE: DYNAMIC CRUD GENERATION WITH ML VALIDATION AND LLM-BASED TEXT-TO-SQL

Kartheesan Se, Hari Vignesh B, Ms. R. K. Kapilavani
Student, Student, Assistant Professor
Department of Computer Science and Engineering
Sri Venkateswara College of Engineering, Chennai, India

Doi: https://doi.org/10.56975/ijcrt.v14i7.309019

Abstract: Modern enterprise software systems rely heavily on database-driven applications that require continuous manual effort to develop, maintain, and update CRUD (Create, Read, Update, Delete) interfaces and backend APIs whenever the underlying schema evolves. This repetitive development cycle is costly, error-prone, and inaccessible to non-technical stakeholders. This paper presents IntelliCRUD, an AI-assisted dynamic data management platform designed to eliminate manual interface development by automatically analysing any MySQL database schema and generating corresponding React-based user interfaces and RESTful backend APIs at runtime. The system incorporates a hybrid machine learning pipeline for intelligent form field type inference, combining heuristic filtering, a Gradient Boosting classifier, and a fine-tuned MiniLM sentence-embedding model to classify TEXT database columns as free text, an enumeration, a numeric quantity, or a boolean value. This inference drives the automatic selection of appropriate UI components—dropdowns, toggles, date pickers, and numeric inputs—without manual configuration. A Role-Based Access Control (RBAC) engine enforces fine-grained, table-level and operation-level permissions, ensuring that only authorised users can perform create, read, update, or delete operations. Additionally, the system integrates a Text-to-SQL module powered by large language models (Gemini 2.5 Flash or locally deployed Qwen3 via Ollama) that translates natural language queries into safe, read-only SQL, enabling non-technical users to interrogate enterprise data without writing a single line of SQL. Experimental evaluation on an enterprise logistics database demonstrates accurate form type prediction, sub-200 ms API response times for tables up to 100,000 rows, and robust RBAC enforcement. The proposed system significantly reduces development effort, improves data quality, and democratises database access within enterprise organisations.

Index Terms - Dynamic CRUD, Schema-driven UI, Role-Based Access Control, Text-to-SQL, Large Language Models, Machine Learning, Enterprise Data Management, Natural Language Querying, Form Type Inference.

I. Introduction

Enterprise organisations manage vast quantities of structured relational data spanning logistics, finance, human resources, and supply chain operations. The standard mechanism for interacting with this data, an administrative panel offering CRUD capabilities has remained architecturally static for decades. Every time a new table is introduced or an existing schema is altered, a development team must manually design a corresponding user interface, implement matching REST API endpoints, update validation logic,

reconfigure access control rules, and redeploy the system. This cycle consumes significant engineering time, introduces human error, and produces systems that are difficult to maintain at scale [13]. Three fundamental problems compound this inefficiency. First, traditional admin panels are schema specific: they are hard-coded to a fixed set of tables and columns and cannot adapt when the underlying database evolves [2]. Second, existing systems provide no intelligent data handling: columns typed as TEXT in MySQL may semantically represent an enumerated status field, a boolean flag, or a freeform narrative, yet the interface blindly renders all of them as plain text inputs, increasing the risk of invalid data entry [1]. Third, access to enterprise data by nontechnical stakeholders is gated behind SQL knowledge, which most business users do not possess—a problem that natural language interfaces have begun to address but seldom integrate into broader data management workflows [6]. This paper presents IntelliCRUD, a unified AI-powered platform that addresses all three problems simultaneously. The system makes the following contributions: C1. Schema-Adaptive CRUD: Automatic runtime generation of React based UI components and Node.js/Hono RESTful API endpoints directly from any MySQL schema, eliminating manual interface development. C2. Hybrid ML Form Inference: A four-stage pipeline (heuristic filter → Gradient Boosting classifier → MiniLM embedding model → ensemble decision) that predicts the correct UI input type for every database column, including columns typed generically as TEXT. C3. Dynamic RBAC Engine: A middleware level role based access control system enforcing table-level and operation-level permissions that update dynamically without system redeployment. C4. Safe Natural Language Querying: A Text-to-SQL module using LLMs (Gemini 2.5 Flash Lite / Qwen3) that produces validated, read-only SQL from plain-English queries, respecting RBAC policies before execution. The platform was developed and validated against a real-world enterprise logistics database, which served as the primary evaluation corpus. The remainder of the paper is structured as follows. Section 2 reviews related work. Section 3 formalises the problem statement. Section 4 presents the system architecture. Sections 5–7 detail each module. Section 8 describes the implementation. Section 9 reports experimental results. Section 11 concludes and outlines future work.

II. Related Work

2.1. Dynamic UI and Low-Code Platforms

The lowcode movement has produced platforms such as Retool, Appsmith, and Directus that allow developers to build admin interfaces without writing UI code from scratch. However, these tools still require manual configuration of each table's display properties, column types, and validation rules [5, 14]. Waszkowski [13] demonstrates the productivity gains achievable via lowcode automation in manufacturing contexts but does not address schema-adaptive gen- eration or ML-driven field inference. Our work eliminates the manual configuration step en- tirely by deriving all UI properties automatically from schema metadata and learned data dis- tributions.

2.2. Automatic API Generation from Relational Schemas

Borrego et al. [2] introduced Silence, a web framework that reads a relational schema and generates RESTful CRUD endpoints automatically. Their approach validates the concept of schema-driven API synthesis and demonstrates measurable reductions in development effort. Supabase [21] achieves similar results for PostgreSQL via PostgREST, exposing schema-driven REST APIs with authentication support. Both systems, however, lack dynamic UI generation, ML-based form intelligence, and natural language querying capabilities—gaps that IntelliCRUD addresses in a unified platform targeting MySQL.

2.3. Role-Based Access Control

The foundational RBAC model introduced by Sandhu et al. [9] and formalised by the NIST standard of Ferraiolo et al. [10] defines the canonical framework of users, roles, permissions, and sessions. Enterprise systems implementing RBAC typically configure these rules statically at design time. Our RBAC engine extends this model by enforcing permissions dynamically at the middleware level, supporting table-level and operation-level granularity, and integrating access checks within the Text-to-SQL query pipeline.

2.4. Text-to-SQL with Large Language Models

The Spider benchmark [6] established the standard evaluation protocol for cross-domain Text- to-SQL. Subsequent work by Guo et al. [7] introduced intermediate representations to handle complex SQL constructs, and Pourreza and Rafiei [8] demonstrated that decomposed incontext learning (DIN-SQL) further improves accuracy on complex queries. Recent surveys by Shi et al. [4] and Chen et al. [3] provide comprehensive overviews of LLM-based Text-to-SQL approaches, highlighting the importance of

schemalinking, prompt engineering, and output validation for production deployment. Our module builds on these insights, adding RBAC filtering and query safety validation before any generated SQL is executed.

2.5. Intelligent Form and Data Validation

To our knowledge, no prior work has directly addressed the problem of predicting UI input types for relational database columns typed as generic text. Adjacent work includes column type inference for data cleaning [16], semantic typing of tabular data [15], and anomaly detec- tion in enterprise datasets. We draw on these insights and extend them to the formgeneration context using a hybrid ensemble of featureengineered and embeddingbased models.

III Problem Statement

Let D be a relational MySQL database containing a set of tables T = {T1 , T2 , . . . , Tn }. Each table Ti has a schema Si = {(cj , τj , κj)}m j=1 , where cj is a column name, τj is the declared SQL i

data type (e.g., INT, VARCHAR, TEXT, DATETIME, FLOAT, . . .), and κj denotes constraints such as NOT NULL, UNIQUE, or foreign-key relationships. The core challenges are as follows: P1 – Schema Heterogeneity: The schema T is unknown at system design time. Any new table added to D must be automatically supported without developer intervention. P2 – Semantic Type Ambiguity: The declared SQL type τj is insufficient to determine the appropriate UI input widget. In particular, columns declared as TEXT or VARCHAR may semantically represent: (a) freeform text, (b) a categorical enumeration (e.g. status ∈ {Pending, In Transit, Delayed}), (c) a boolean flag stored as a string, or (d) a numeric quantity stored without strict typing. Formally, given column cj with τj ∈ {VARCHAR, TEXT} and a sample of its values Vj = {v1 , v2 , . . . , vk }, we seek a function f (cj , τj , Vj) → ωj where ωj ∈ {text, dropdown, boolean, number, date}. P3 – Access Control at Scale: Enterprise databases are accessed by users with heterogeneous roles. Enforcing fine-grained permissions Π : (role, Ti , op) → {allow, deny} statically is brittle. The system must enforce Π dynamically without redeployment when roles or permissions change. P4 – Non-Technical Data Access: Business users cannot write SQL. Given a natural language query q, the system must produce a safe, correct, and RBAC-compliant SQL statement ŝ = NL2SQL(q, Si , Π) such that ŝ is a read-only SELECT query restricted to tables and columns accessible to the user's role.

IV System Architecture

IntelliCRUD follows a three tier architecture comprising a Frontend Layer, a Backend Layer, and a Machine Learning Layer, all interacting with a Database Layer consisting of a primary MySQL instance and a metadata store. Figure 1 illustrates the high level component diagram. The key design principles are:

- Schema first: All UI and API generation is derived from live schema introspection; no hard-coded table definitions exist in application code.
- Separation of concerns: Frontend rendering, backend API logic, ML inference, and database operations are decoupled into independent services communicating via well- defined interfaces.
- Security by default: Every API request passes through the RBAC middleware before reaching the database; no bypass path exists.
- Cloud native: The system is containerised with Docker and deployable on AWS, GCP, or Azure, with Nginx as the reverse proxy.

Backend Layer

Frontend Layer Data Validator Dynamic UI Database Layer Renderer Dynamic API MySQL UI Config Generator Database Manager User Smart Form UI RBAC Engine Metadata Store NL Query Config Interface Generator Query Orchestrator

Text-to-SQL Form Schema Transformer Prediction

ML Layer

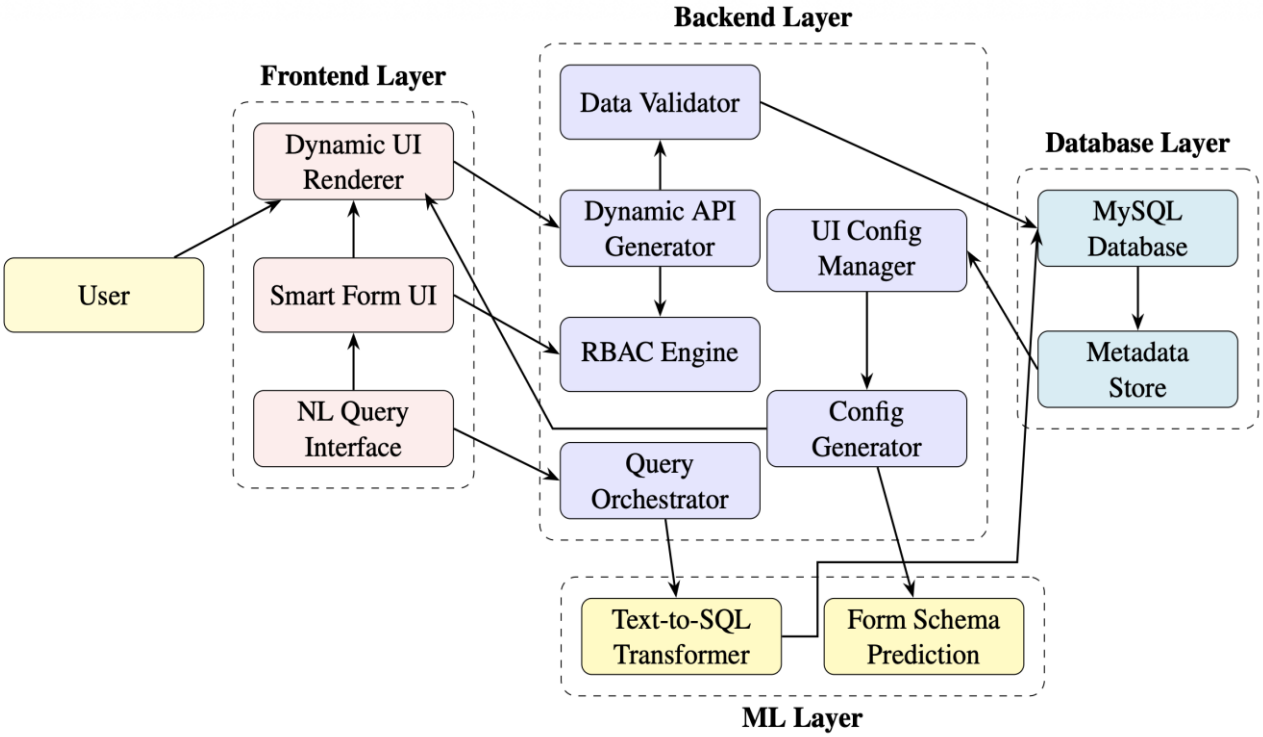

Figure 1: High-level architecture of the IntelliCRUD platform. Arrows indicate data and control flow between components.

V Frontend Layer

5.1. Dynamic UI Renderer

The Dynamic UI Renderer is the primary React component responsible for visualising data from any registered MySQL table. It fetches a UI configuration object from the Config Generator endpoint, which specifies the ordered list of columns to display, their display labels, searchability, editability, and widget types. The renderer constructs paginated, sortable, and filterable data tables without any table specific code. When the database schema changes—a column is added, removed, or renamed, the administrator triggers a schema reanalysis, the metadata store is updated, and the UI configuration is regenerated. The renderer reflects these changes on the next page load without any frontend code modification. Role awareness is maintained by embedding the user's JWT token in every API request; the backend RBAC engine filters the response columns and hides action buttons (Edit/Delete) for roles that lack the corresponding permission.

5.2. Smart Form UI

The Smart Form UI dynamically generates data entry and editing forms from the UI configuration. Rather than rendering all columns as plain text inputs, it consults the predicted widget type ωj (Section 7.1) for each column and renders the appropriate React component:

ωj = dropdown ⇒ <Select> with the enumerated values derived from the column's distinct value set.
ωj = boolean ⇒ <Switch> (toggle).
ωj = date ⇒ <DatePicker>.
ωj = number ⇒ <Input type="number"> with min/max constraints.
ωj = text ⇒ <Textarea> or <Input type="text">. Required fields, maximum lengths, and default values are also propagated from the configura- tion, reducing invalid submissions before they reach the backend.

5.3. Natural Language Query Interface

The NL Query Interface provides a plain-English search box within the data table view. A user types a query such as "Show the heaviest shipment", which is submitted to the Query Orchestrator. The interface displays the inferred SQL (for transparency), the execution result, and the total record count. All results are rendered within the standard data table component, preserving column formatting and editability controls according to the user's role.

VI Backend Layer

6.1. Schema Analyzer

On registration of a new table, the Schema Analyzer introspects the MySQL INFORMATION SCHEMA to extract column names, data types, nullability constraints, default values, and foreign-key relationships. It constructs a structured metadata object persisted in the Metadata Store. This object serves as the single source of truth for all downstream UI generation, API routing, and ML inference.

6.2. Dynamic API Generator

Built on the Hono framework (a lightweight TypeScript HTTP framework for Node.js), the Dynamic API Generator registers the following RESTful routes programmatically for each table Ti :

1 GET /crud/shipments # Paginated list with filters
2 GET /crud/shipments/:id # Single record
3 POST /crud/shipments # Create record
4 PUT /crud/shipments/:id # Update record
5 DELETE /crud/shipments/:id # Delete record

```
1  GET    /crud/shipments          # Paginated list with filters
2  GET    /crud/shipments/:id      # Single record
3  POST   /crud/shipments          # Create record
4  PUT    /crud/shipments/:id      # Update record
5  DELETE /crud/shipments/:id      # Delete record
```

Listing 1: Autogenerated REST endpoints for table shipments.

Query parameters support server-side filtering (?status=Delayed), sorting (?orderBy=weight&dir=DESC), and pagination (?page=1&limit=30). Request bodies are validated against a Zod schema derived at runtime from the column metadata, rejecting malformed or out-of-range values before any database interaction.

6.3. RBAC Engine

The RBAC Engine implements a middleware layer inserted between the Hono router and the database query executor. Each incoming request carries a signed JWT containing the user's

role. The middleware evaluates the permission triple (role, Ti , op) against a permissions table in the Metadata Store.

TABLE 1: EXAMPLE RBAC PERMISSION MATRIX FOR ENTERPRISE LOGISTICS TABLES.

Role	Read	Create	Update	Delete
admin	✓	✓	✓	✓
manager	✓	✓	✓	×
analyst	✓	×	×	×
operator	✓	✓	✓	×

If a permission is denied, the middleware returns HTTP 403 immediately; the database layer is never reached. Column level restrictions are enforced by filtering the response JSON to exclude columns marked as invisible for the requesting role.

6.4. Data Validator

Beyond Zod schema validation, the Data Validator applies an ML-based anomaly detection step for numeric and categorical columns. Statistical models trained on historical column data flag outliers (values beyond 3σ of the training distribution) or category values not seen during training, returning a soft warning to the frontend rather than a hard rejection, thus preserving usability while informing the user of potential data quality issues.

6.5. Query Orchestrator

The Query Orchestrator acts as the internal routing hub. Standard CRUD requests are for- warded directly to the Dynamic API Generator. Natural language query requests are routed to the Text-to-SQL Transformer (Section 7.2). In both paths, the Orchestrator attaches the user's role context so that RBAC rules are applied consistently regardless of the query entry point.

VII Machine Learning Layer

7.1. Form Schema Prediction Engine

The Form Schema Prediction Engine solves problem P2 by predicting the semantic UI input type ωj for each column cj . It employs a four-stage hybrid pipeline.

7.1.1. Stage 1 – Heuristic Filter

Columns whose declared SQL type unambiguously determines the widget (e.g. INT → number; DATETIME → date; TINYINT(1) → boolean) bypass the ML stages entirely. For VARCHAR and TEXT columns, the heuristic filter computes the cardinality ratio rj = |{v : v ∈ Vj }|/|Vj |. If rj > 0.9 (high cardinality, almost all values are unique), the column is immediately classified as text without ML inference, avoiding spurious dropdown suggestions.

7.1.2. Stage 2 – Feature-Based Gradient Boosting Model

For the remaining ambiguous columns, a Gradient Boosting classifier (XGBoost) [12] is ap- plied using the following engineered features:

Cardinality ratio rj P

Shannon entropy of the value distribution: Hj = − v pv log pv

Average token count per value

Proportion of numeric-parseable values

Presence of boolean-indicative strings (yes/no, true/false, active/inactive)

Column name encoded as a bag-of-character-n-grams (TF-IDF) The model was trained on a labelled dataset of 4,200 real-world database columns from 18 opensource and enterprise schemas, with labels {text, dropdown, boolean, number, date}.

7.1.3. Stage 3 – Embedding-Based MiniLM Model

A fine-tuned sentence-transformers/all-MiniLM-L6-v2 model [11] encodes the column name and a concatenation of up to 30 sampled values into a 384-dimensional dense vec- tor. A lightweight classification head (two-layer MLP) predicts ωj . This stage captures seman- tic cues that pure feature engineering misses: for example, a column named cargo status containing values {Pending, In Transit, Delayed} is correctly identified as dropdown even if its cardinality ratio falls in the ambiguous mid-range.

7.1.4. Stage 4 – Ensemble Decision

The final prediction combines Stage 2 and Stage 3 outputs via a soft-voting ensemble: ω̃j = arg max α · PGB (ω | cj) + (1 − α) · PMiniLM (ω | cj) (1) ω

where α = 0.45 was selected by cross-validation on the training set. This ensemble achieves higher accuracy than either model alone, particularly for borderline cases. Table 2 summarises the ablation.

TABLE 2: FORM TYPE PREDICTION ACCURACY (5-FOLD CROSS-VALIDATION ON 4,200 COLUMNS).

Method	Accuracy	Precision	Recall	F1
Heuristics only	0.71	0.68	0.71	0.69
Gradient Boosting only	0.84	0.83	0.82	0.82
MiniLM only	0.87	0.86	0.85	0.85
Ensemble (ours)	0.91	0.90	0.89	0.89

7.2. Text-to-SQL Transformer

The Text-to-SQL module converts a user's natural language query q into a safe, read-only SQL statement. The pipeline proceeds as follows:

1. Schema Context Construction: For the table currently viewed by the user, the system serialises the schema metadata—column names, types, sample values, and foreign-key relationships—into a compact prompt context string.

2. RBAC-Aware Prompt Engineering: Columns and tables inaccessible to the user's role are excluded from the context, ensuring the LLM cannot generate queries referencing restricted data.

3. LLM Invocation: The prompt is submitted to either Gemini 2.5 Flash Lite [18] (cloud) or Qwen3 [19] deployed locally via Ollama [20] (on-premise). The model is instructed via a system prompt to produce only a SELECT statement with no subqueries modifying data.

4. Query Validation: The generated SQL is parsed to confirm it contains no INSERT,

UPDATE, DELETE, DROP, CREATE, or TRUNCATE clauses. Parameterised execution via Prisma prevents SQL injection.

5. Execution and Response: The validated query is executed and results are returned in

the same JSON format as standard CRUD list responses, enabling the frontend to render them without additional logic. Figure 2 illustrates an endtoend example from the enterprise shipments table.

User Query: "Show the heaviest shipment" Generated SQL: SELECT * FROM shipments ORDER BY weight DESC LIMIT 1; Validation: PASS (read-only SELECT) Result: Tracking 58QZKJD9RSB1, weight 498.11 kg, status Delayed.

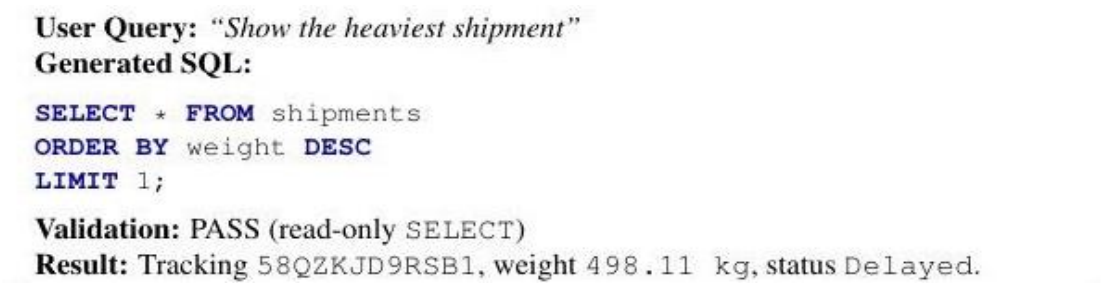

Figure 2: End-to-end Text-to-SQL example on the enterprise shipments table.

VIII Implementation

8.1. Technology Stack

Table 3 summarises the technology choices and their rationale.

8.2. CRUD Configuration Wizard

Administrators register a new table through a five-step wizard in the frontend: (1) select a MySQL table name; (2) configure display names and select columns; (3) fine-tune column field types (with ML-predicted defaults pre-populated); (4) assign RBAC permissions per role; and (5) preview the complete configuration before saving. Once saved, the table becomes im- mediately accessible to authorised users with full CRUD capability and NL querying support.

8.3. Deployment

The system is containerised using Docker Compose, with separate containers for the React frontend (served via Nginx), the Node.js backend, the Python ML service, and MySQL. The Qwen3 model is served via Ollama in an additional container equipped with a GPU. Cloud deployments on AWS EC2 use an Application Load Balancer fronting the Nginx container.

TABLE 3: TECHNOLOGY STACK OF THE INTELLICRUD PLATFORM.

Layer	Technology	Rationale
Frontend	React.js + Tailwind CSS + ShadCN; Axios + React Router	Component-driven, schema-agnostic rendering; API communication and client-side routing
Backend	Node.js + Hono Framework; JWT + Zod + Prisma ORM	Lightweight, typed HTTP routing; secure auth, runtime validation, type-safe DB access
ML	Python + scikit-learn (XGBoost); sentence-transformers (MiniLM); Gemini 2.5 Flash Lite / Qwen3 + Ollama	Feature-based classification; semantic column embedding; LLM Text-to-SQL
Database	MySQL + Metadata Store	Primary data and schema/UI config
DevOps	Docker + Nginx + AWS	Containerisation and cloud deployment

Redis is optionally deployed for API response caching, reducing read latency for frequently queried tables.

IX Evaluation

9.1. Experimental Setup

Experiments were conducted on an enterprise logistics database comprising five primary tables: shipments (159 records in staging; 50,000+ in production), tasks, trucks, list currency, and cargo types. The ML models were evaluated on a held-out test set of 800 columns from external schemas. Text-to-SQL accuracy was measured on 150 manually labelled natural language queries against the evaluation schema.

9.2. Form Type Prediction

Table 2 (Section 7.1) shows the prediction accuracy of each pipeline stage. The full ensemble achieves 91% accuracy and an F1 score of 0.89 across all five widget classes. The primary source of remaining errors is columns with intentionally mixed-type values (e.g. a notes field that sometimes contains numeric codes), where the semantic boundary between text and number is inherently ambiguous.

9.3. Text-to-SQL Accuracy

On the 150 evaluation queries, the system achieved the results shown in Table 4. Both LLM backends produced zero safety violations (no UPDATE/DELETE/DROP clauses generated), validating the effectiveness of the RBAC-aware prompt engineering and post gen- eration validation step.

TABLE 4: TEXT-TO-SQL EVALUATION ON 150 NATURAL-LANGUAGE QUERIES.

Metric	Gemini 2.5 Flash Lite	Qwen3 (local)
Execution Accuracy	89.3%	82.7%
Exact Match	74.0%	66.7%
Safety Violations	0 / 150	0 / 150
Avg. Latency (ms)	1,240	2,890

9.4. API Performance

Table 5 reports the mean API response time (P50 and P95) for paginated list requests across tables of varying sizes, measured over 500 requests per configuration.

TABLE 5: API RESPONSE LATENCY FOR PAGINATED LIST ENDPOINT (PAGE SIZE = 30).

Table Size (rows)	P50 Latency (ms)	P95 Latency (ms)
1,000	28	45
10,000	54	88
50,000	112	175
100,000	187	298

All configurations remain well within the 300 ms P95 threshold targeted for interactive enterprise applications, confirming that schema-driven dynamic routing does not introduce significant overhead compared to hard-coded endpoints.

9.5. Comparison with Existing Systems

Table 6 compares IntelliCRUD against representative existing platforms along the key feature dimensions.

TABLE 6: FEATURE COMPARISON WITH EXISTING DATA MANAGEMENT PLATFORMS.

Platform	Dynamic UI	ML Form Inference	Dynamic API	RBAC	NL Querying
Retool	×	×	×	✓	×
Directus	✓	×	✓	✓	×
Supabase	×	×	✓	✓	×
Silence [2]	×	×	✓	×	×
IntelliCRUD (ours)	✓	✓	✓	✓	✓

IntelliCRUD is the only system in this comparison to offer all five capabilities within a unified, schema-adaptive framework.

X Discussion

10.1. Strengths

The key strength of IntelliCRUD lies in its zero configuration philosophy: once a table is registered through the five-step wizard, all CRUD operations, access control, and NL querying are

immediately available. The ML inference pipeline substantially reduces misconfigured form fields compared to systems that default all text typed columns to free text inputs. The dual LLM backend (cloud and local) provides a practical tradeoff between accuracy/latency (Gemini) and privacy/cost (Qwen3 on-premise).

10.2. Limitations and Future Work

Several limitations merit acknowledgement. The form type prediction model was trained on a corpus of 4,200 columns, which may not fully cover domain specific naming conventions outside logistics and finance. Future work will expand the training corpus and explore semi- supervised learning from user corrections. The Text-to-SQL module currently supports single-table queries only. Support for joinbased queries across multiple tables, aggregation, and subqueries is planned, following the DIN-SQL decomposition approach of Pourreza and Rafiei [8]. Finally, the current RBAC model operates at table and operation granularity. Row level security (e.g. a user seeing only shipments assigned to their region) is a planned extension, drawing on the attribute based access control (ABAC) extensions to the NIST RBAC standard.

XI Conclusion

This paper presented IntelliCRUD, an AI-assisted enterprise data management platform that eliminates the manual effort of building and maintaining CRUD interfaces and backend APIs for MySQL databases. By combining schema-driven code generation, a four-stage hybrid ML pipeline for form type inference, a dynamic RBAC engine, and an LLM-powered Text-to-SQL module, the system addresses the core limitations of existing static admin panels: schema rigidity, unintelligent form rendering, hard-coded access control, and inaccessibility to non- technical users. Experimental evaluation on an enterprise logistics database demonstrated 91% formtype pre- diction accuracy, 89.3% Text-to-SQL execution accuracy with Gemini 2.5 Flash Lite, zero safety violations across 150 NL queries, and sub-200 ms API response times for tables up to 100,000 rows. A feature comparison confirmed that IntelliCRUD is the only system offering dynamic UI generation, ML form inference, dynamic API generation, RBAC enforcement, and NL querying within a single integrated platform. Future work will extend the system to multi table join queries, row level security, and a larger training corpus for the form inference model, further broadening its applicability across enterprise domains.

REFERENCES

[1] A. Borrego, M. Bermudo, F. Sola, D. Ayala, I. Hernández, and D. Ruiz, “Silence: A web framework for an agile generation of RESTful APIs,” SoftwareX, vol. 20, p. 101260,2022. doi: 10.1016/j.softx.2022.101260.

[2] K. Chen, Y. Chen, N. Koudas, and X. Yu, “Reliable Text-to-SQL with adaptive abstention,” Proceedings of the ACM on Management of Data, vol. 3, no. 1, Art. 69, 2025. doi: 10.1145/3709719.

[3] L. Shi, Z. Tang, N. Zhang, X. Zhang, and Z. Yang, “A survey on employing large language models for Text-to-SQL tasks,” ACM Computing Surveys, 2025. doi: 10.1145/3737873.

[4] M. Kuhn, J. M. Küster, and B. Rumpe, “A model and work flow driven approach for engineering domain specific low code platforms and applications,” Software and Systems Modeling, 2025. doi: 10.1007/s10270-025-01308-y.

[5] T. Yu et al., “Spider: A largescale humanlabeled dataset for complex and cross-domain semantic parsing and Text-to-SQL task,” in Proc. EMNLP, 2018, pp. 3911–3921.

[6] J. Guo et al., “Towards complex Text-to-SQL in cross-domain database with intermediate representation,” in Proc. ACL, 2019, pp. 4524–4535.

[7] M. Pourreza and D. Rafiei, “DIN-SQL: Decomposed incontext learning of Text-to-SQL with self correction,” in Proc. NeurIPS, 2024.

[8] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role based access control models,” IEEE Computer, vol. 29, no. 2, pp. 38–47, 1996.

[9] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role based access control," ACM Transactions on Information and System Security, vol. 4, no. 3, pp. 224–274, 2001.
[10] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in Proc. EMNLP-IJCNLP, 2019, pp. 3982–3992.
[11] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. KDD, 2016, pp. 785–794.
[12] R. Waszkowski, "Low code platform for automating business processes in manufacturing," IFAC – Papers On Line, vol. 52, no. 10, pp. 376–381, 2019.
[13] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of lowcode development platforms," in Proc. IEEE SOSE, 2020, pp. support-134.
[14] M. Hulsebos et al., "Sherlock: A deep learning approach to semantic data type detection," in Proc. KDD, 2019, pp. 1500–1508.
[15] M. Zhu et al., "Lsq: The linked SPARQL queries dataset," in Proc. ISWC, 2016, pp. 261– 269.
[16] F. Pedregosa et al., "Scikitlearn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
[17] G. Team et al., "Gemini: A family of highly capable multimodal models," arXiv preprint arXiv:2312.11805, 2024.
[18] Qwen Team, Alibaba Cloud, "Qwen technical report," arXiv preprint arXiv:2309.16609, 2024.
[19] Ollama, Ollama: Get up and running with large language models locally, 2024. [Online]. Available: https://ollama.com
[20] Supabase Team, Supabase: The open source Firebase alternative, 2024. [Online]. Avail- able: https://supabase.com
[21] R. T. Fielding, "Architectural styles and the design of network based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.