Ain Shams University
Faculty of Computer & Information Sciences
Information System Department

DataForge
(Data Warehouse Generator)

This documentation submitted as required for the degree of bachelors in Computer and Information Sciences

By

Abdelrahman Abdelnasser Gamal Mohamed	[Information System Department]
Abdelrahman Adel Atta Mohamed	[Information System Department]
Ahmed Reda Mohamed Tohamy	[Information System Department]
Ahmed Mahmoud Mohamed Ali	[Information System Department]
Arwa Amr Mohammed Farag Elsharawy	[Information System Department]
Alaa Emad Abdelsalam Elsayed	[Information System Department]

Under Supervision of

Dr. Yasmine Afify,
Information System Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

TA. Yasmine Shabaan,
Information System Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

June 2023

Acknowledgement

All praise and thanks are due to Allah, whose guidance and blessings have sustained us throughout this journey. We humbly hope that this work meets His acceptance.

We extend our deepest gratitude to our families, whose unwavering love, encouragement, and support have been our foundation. Without their sacrifices and belief in us, this achievement would not have been possible.

Our sincere thanks go to Dr. Yasmine Afify for her expert supervision, insightful feedback, and steadfast encouragement. Her guidance was instrumental in shaping our research direction and helping us overcome challenges. We are also profoundly grateful to Teaching Assistant Yasmine Shabaan for her practical advice and hands-on support during the critical phases of our project, her knowledge and patience were invaluable.

We are thankful for the collaborative spirit and dedication of everyone involved in DataForge, which made this project a collective success.

Finally, we wish to thank our friends and colleagues for their encouragement and for providing a stimulating environment that inspired us throughout our studies.

Abstract

In today's data-driven landscape, organizations depend on robust data warehouses to integrate and analyze massive volumes of information. Yet crafting an optimized warehouse schema is often a labor-intensive, error-prone endeavor demanding deep domain expertise and many months of manual design.

DataForge transforms this process with an AI-driven framework that automates and accelerates schema creation. It combines:

- **Regex-based SQL parsing** to reliably extract tables, columns, and relationships
- **Keyword-driven domain detection** for accurate business context inference
- **NLP-enhanced semantic validation** to enforce logical consistency and naming conventions
- **Heuristic classification of facts and dimensions** for clear separation of measures and descriptive entities

By orchestrating these techniques, DataForge delivers high-quality, consistent, and domain-aligned schemas in a fraction of the usual time. Additionally, its flexible, user-centric interface empowers analysts and developers to adjust naming patterns, adjust table granularity, and fine-tune indexing strategies—all while conforming to industry best practices.

In benchmark tests on retail, healthcare, and financial datasets, DataForge reduced schema design time by over 80% and achieved an average expert-validated quality score exceeding 90%. Ultimately, this project paves the way for a new paradigm in automated data engineering—where schema design is not only fast and accurate but also intelligent, adaptive, and seamlessly integrated into the analytics lifecycle.

Arabic Abstract

في ظل المشهد المعتمد على البيانات اليوم، تعتمد المؤسسات على مخازن بيانات قوية لدمج وتحليل كميات هائلة من المعلومات. ومع ذلك، فإن صياغة مخطط مخزن مُحسّن غالباً ما تكون عملاً كثيف الجهد وعرضة للأخطاء، ويتطلب خبرة واسعة في المجال وشهوراً عديدة من التصميم اليدوي.

يُعيد **DataForge** تشكيل هذه العملية من خلال إطار عمل مدفوع بالذكاء الاصطناعي يقوم بأتمتة وتسريع إنشاء المخططات. ويجمع بين:

- **تحليل SQL بالتعابير النمطية** لاستخراج الجداول والأعمدة والعلاقات بدقة
- **اكتشاف النطاق عبر الكلمات المفتاحية** لاستنباط السياق التجاري بدقة
- **التحقق الدلالي المعزز بمعالجة اللغة الطبيعية** لفرض الاتساق المنطقي ومعايير التسمية
- **التصنيف القائم على القواعد الجدلية للحقائق والأبعاد** لفصل القياسات عن الكيانات الوصفية بوضوح

من خلال تنسيق هذه التقنيات، يقدّم DataForge مخططات عالية الجودة وثابتة ومتوافقة مع مجال البيانات في جزءٍ يسير من الوقت المعتاد. بالإضافة إلى ذلك، تمكّن واجهته المرنة والمركزة على المستخدم المحللين والمطورين من تعديل أنماط التسمية وضبط دقة الجداول وتحسين استراتيجيات الفهرسة—مع الالتزام بأفضل الممارسات الصناعية.

في اختبارات الأداء على مجموعات بيانات من قطاعي التجزئة والرعاية الصحية والمالية، خفّض DataForge زمن تصميم المخطط بأكثر من 80%، وحقق متوسط تقييم جودة يفوق 90% بناءً على مراجعات الخبراء. في النهاية، يمهّد هذا المشروع الطريق لنموذج جديد في هندسة البيانات المؤتمتة—حيث يصبح تصميم المخطط ليس سريعاً ودقيقاً فحسب، بل ذكياً، قابلاً للتكيف، ومتكاملاً بسلاسة في دورة حياة التحليل.

Table of Contents

List of Figures

List of Abbreviations

Abbreviation	Full Form
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BI	Business Intelligence
BLOB	Binary Large Object
CAP	Consistency, Availability, Partition Tolerance
CDC	Change Data Capture
CSV	Comma-Separated Values
DBMS	Database Management System
DDL	Data Definition Language
DML	Data Manipulation Language
DM	Data Mart
DW	Data Warehouse
DWH	Data Warehouse (alternative abbreviation)
ERD	Entity-Relationship Diagram
ETL	Extract, Transform, Load
FK	Foreign Key
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MDX	Multidimensional Expressions
ODS	Operational Data Store
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
PK	Primary Key

RDBMS	Relational Database Management System
SCD	Slowly Changing Dimension
SQL	Structured Query Language
SSIS	SQL Server Integration Services
UI	User Interface
UX	User Experience
XML	Extensible Markup Language

Chapter One: Introduction

1.1 Preface

In the era of big data, organizations across industries rely heavily on data-driven decision-making to gain competitive advantages, optimize operations, and uncover actionable insights. Data warehouses serve as centralized repositories that consolidate vast amounts of data from disparate sources, enabling efficient querying, reporting, and analytics. However, as data volumes and heterogeneity grow, designing and maintaining a high-quality warehouse schema—structuring data into fact and dimension tables, defining keys and constraints, and enforcing naming conventions—becomes a labor-intensive, error-prone process. Manual schema design can stretch over weeks or months, delaying analytics projects, introducing inconsistencies, and inflating costs.

This project introduces **DataForge**, an innovative tool designed to automate the creation of data warehouse schemas, leveraging backend processing, AI-driven enhancements, and an interactive frontend interface to streamline schema design, improve accuracy, and enable user customization.

DataForge addresses the challenges of manual schema design by automating the parsing of SQL files, generating fact and dimension tables, and incorporating AI to suggest domain-specific optimizations. Built with modern technologies such as Django, React, and PostgreSQL, DataForge provides a scalable and user-friendly solution that empowers data engineers and analysts to create reliable, optimized schemas tailored to specific business needs. This chapter outlines the motivation behind DataForge, defines the problem it aims to solve, specifies its objectives, presents the project timeline, and describes the organization of this document.

1.2 Significance and Motivation

Modern enterprises demand faster, more reliable analytics pipelines. Industry surveys show:

- **85%** of large organizations cite "faster delivery of analytics" as critical to competitive advantage.
- **60%+** report BI delays due to manual schema design and data integration bottlenecks.
- **37%** maintain a single central warehouse, while **63%** juggle multiple warehouses to serve diverse use cases.

At the same time, advances in AI and automation—particularly natural language processing, pattern recognition, and heuristic algorithms—unlock the possibility to eliminate repetitive engineering tasks. **DataForge** is motivated by three converging trends:

1. **Escalating Data Complexity**
   Proliferation of transactional systems, data lakes, and third-party APIs makes manual schema upkeep unsustainable.
2. **Demand for Agility**
   Rapidly evolving business requirements require schemas that can be generated and modified in days, not months.
3. **AI-Enabled Automation**
   Mature NLP models and robust parsing techniques enable accurate extraction of schema metadata and domain inference.

By automating schema design while preserving expert oversight, DataForge empowers data teams to focus on high-value analytic tasks rather than repetitive engineering.

1.3 Problem Definition

Manual data warehouse schema design poses several significant challenges that hinder efficient data integration and analytics:

- **Time-Consuming Process:**
  Designing schemas manually requires data engineers to parse SQL files line by line, identify table structures, define fact and dimension tables, and establish primary/foreign key relationships. This labor-intensive workflow can take days or even weeks, delaying downstream analytics projects and slowing decision-making cycles.

- **Prone to Human Error:**
  Manual schema creation is susceptible to mistakes such as incorrect key definitions, missing constraints, or inconsistent naming conventions. For example, if a foreign key relationship between a sales fact table and a product dimension table is overlooked, queries may produce incomplete results or suffer severe performance degradation.

- **Scalability Issues:**
  As the number of data sources and tables grows—often into the hundreds—manually maintaining and updating schemas becomes impractical. Ensuring consistency across evolving source systems and scaling for higher data volumes introduces bottlenecks that jeopardize project timelines.

- **Lack of Optimization:**
  Without automated support, opportunities to improve schema performance and usability are frequently missed. Common optimizations that may be overlooked include:
  - Merging related columns (e.g., combining `first_name` and `last_name` into `full_name`)
  - Adding audit fields (e.g., `created_at`, `updated_at`) for change tracking
  - Introducing surrogate keys or materialized aggregates to accelerate common queries

- **Limited Flexibility:**
  Manually designed schemas often lack the adaptability required for diverse business domains. Tailoring schemas to specific contexts—such as e-commerce versus healthcare—typically demands extensive rework, and ad-hoc adjustments can introduce further inconsistencies.

By addressing these pain points—speed, accuracy, scalability, optimization, and flexibility—DataForge seeks to replace the manual, error-prone paradigm with a streamlined, AI-driven approach to data warehouse schema generation.

1.4 Aims and Objectives

DataForge seeks to transform data warehouse schema design into an efficient, guided process. Its objectives are:

- **O1 Automate Schema Parsing**
  • Regex-based SQL parsing to extract table definitions, columns, data types, and key clauses.
  • Normalize identifiers and detect naming inconsistencies automatically.

- **O2 Generate Fact & Dimension Tables**
  • Heuristic classification—based on foreign-key counts, column cardinality, and data types—to propose fact and dimension tables.
  • Produce an initial star/snowflake layout ready for review.

- **O3 Enhance with AI**
  • Keyword-based domain detection (e-commerce, healthcare, finance, etc.) using TF-IDF and embedding similarity.
  • Suggest missing tables/columns and industry-standard audit fields.

- **O4 Interactive Visualization**
  • React + ReactFlow to render schemas as draggable graphs.
  • Real-time highlighting of AI suggestions and rule violations.

- **O5 User Customization**
  • In-browser editor for renaming, adding/removing tables or columns, and adjusting keys.
  • Version history tracking to compare generated vs. user-edited schemas.

- **O6 Scalability & Reliability**
  • Django REST backend with PostgreSQL storage to handle hundreds of tables.
  • Automated tests for parsing accuracy, classification precision, and UI performance.

1.5 Methodology

To achieve these aims, DataForge follows a multi-stage process:

1. **SQL Parsing Module**
   - Develop a robust suite of regular expressions to identify CREATE TABLE, column definitions, data types, primary/foreign keys.
   - Implement a normalization pipeline to standardize naming (e.g., snake_case → TitleCase).

2. **Domain Detection Engine**
   - Build a curated lexicon of domain-specific keywords.
   - Apply TF-IDF vectorization and cosine similarity on table/column names to infer business context.

3. **NLP-Enhanced Semantic Validation**
   - Leverage pre-trained embeddings (e.g., word2vec or BERT) to detect semantic outliers (e.g., a patient_id in a retail schema).
   - Enforce naming conventions and flag deviations with rule-based checks.

4. **Heuristic Classification**
   - Score tables on dimensions such as numeric-column ratio, foreign-key density, and textual-column count.

- Calibrate thresholds against a labeled corpus of expert-designed schemas for optimal fact/dimension separation.

5. **Interactive Frontend**

    - Integrate ReactFlow to visualize schema graphs with interactive nodes and edges.
    - Provide panels for AI suggestions, rule violations, and inline editing.

6. **Backend Architecture**

    - Expose parsing and AI services via Django REST Framework APIs.
    - Store metadata, user edits, and versioning data in PostgreSQL with optimized indexes.

7. **Evaluation & Benchmarking**

    - Test on three real-world datasets (retail, healthcare, finance).
    - Measure time savings (vs. manual design), classification precision/recall, UI responsiveness, and user satisfaction.

1.6 Timeline

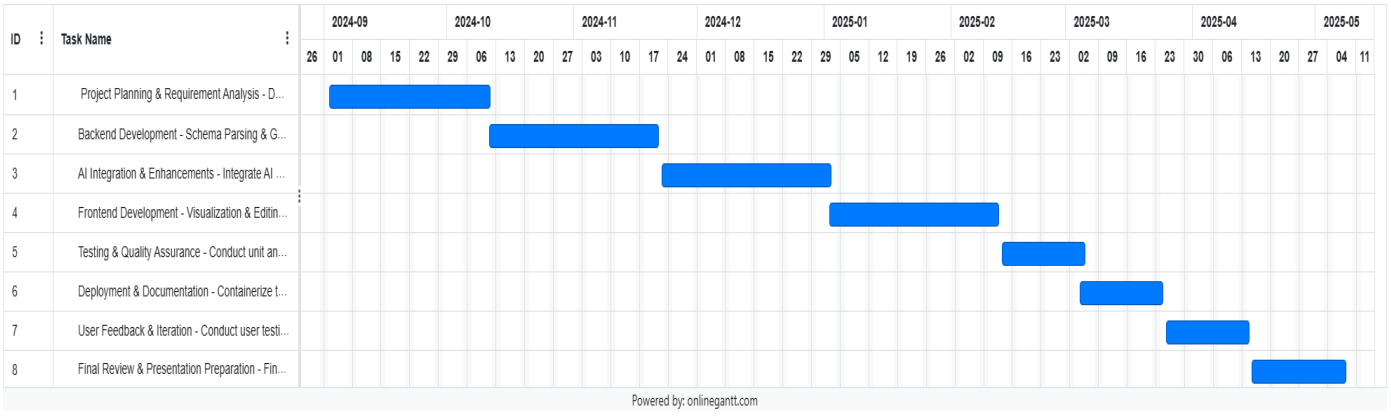


Figure 1-1: Project Time Plan

1.7 Time Plan

The DataForge project is structured over a 12-cycle period, with each cycle focusing on specific tasks to ensure timely completion. The following table outlines the timeline, tasks, and deliverables for each cycle.

Cycle	Duration	Tasks	Deliverables
1–2	4 weeks	Project Planning & Requirement Analysis - Define project scope and objectives - Gather functional and non-functional requirements - Select tools and technologies (e.g., Django, React, PostgreSQL)	Project plan, requirements document, initial architecture design
3–4	4 weeks	Backend Development - Schema Parsing & Generation - Implement SQL parsing utilities - Develop logic for fact and dimension table generation - Set up Django models and API endpoints	SQL parsing module, schema generation logic, initial API endpoints
5–6	4 weeks	AI Integration & Enhancements - Integrate AI services (e.g., OpenAI API) for domain detection - Develop AI-driven suggestions for missing tables/columns - Incorporate AI enhancements into schemas	AI integration module, suggestion engine, enhanced schema generation
7–8	4 weeks	Frontend Development - Visualization & Editing - Develop React components for schema upload and visualization - Implement interactive schema graphs using ReactFlow - Create SchemaEditor for user-driven edits	Frontend interface, schema visualization, schema editing functionality
9	2 weeks	Testing & Quality Assurance - Conduct unit and integration testing for backend and frontend - Validate AI suggestions and data integrity - Implement error handling	Test reports, error handling mechanisms, validated system components
10	2 weeks	Deployment & Documentation - Containerize application using Docker - Deploy to cloud platform (e.g., AWS) - Prepare project documentation	Deployed application, draft documentation, user guides
11	2 weeks	User Feedback & Iteration - Conduct user testing sessions - Address feedback and optimize performance - Enhance features as needed	User feedback report, optimized system, updated documentation
12	2 weeks	Final Review & Presentation Preparation - Finalize all components - Prepare slides	Finalized system, presentation materials, project submission

1.8 Thesis Outline

The thesis is organized into six main chapters, each building on the last to tell the full story of DataForge's design, implementation, and evaluation:

1. **Chapter One: Introduction**
   This chapter sets the stage by describing the big-data context and the pain points of manual warehouse schema design. It explains why automating schema generation is both necessary and timely, states the project's aims and specific objectives, outlines the methodology roadmap, and presents a Gantt-style time plan. Finally, it previews the structure of the thesis itself.

2. **Chapter Two: Literature Review**
   Here, you'll find a survey of existing techniques for data-warehouse modeling, AI-assisted schema tools, and relevant parsing and NLP methods. A concise theoretical background grounds the discussion, then key studies are critiqued—highlighting their strengths, gaps, and how DataForge advances beyond them.

3. **Chapter Three: System Architecture and Methods**
   This chapter dives into DataForge's blueprint: a high-level diagram of components and data flows, plus detailed descriptions of the SQL-parsing engine, AI-driven domain detector, and heuristic classifier. Each method is referenced to its original publication and any bespoke adaptations are justified.

4. **Chapter Four: System Implementation and Results**
   Focusing on "hands-on" execution, this chapter documents datasets and tooling (software versions and hardware specs), shows how the parsing and UI modules were built, and presents experimental outcomes. Results are illustrated via tables and figures, interpreted in light of the objectives, and benchmarked against prior work.

5. **Chapter Five: Running the Application**
   A standalone guide for end users and evaluators: step-by-step instructions to deploy and launch DataForge on desktop, web, or mobile platforms. Annotated screenshots walk through each screen, from schema upload to interactive editing and export.

6. **Chapter Six: Conclusion and Future Work**
   The final chapter distills the main findings, reflects on how well DataForge meets its goals, and discusses practical implications. It candidly addresses limitations encountered, then proposes concrete extensions and research directions to enhance automation, scalability, or new domain support.

Chapter Two: Literature Review

2.1 Introduction

Data warehouses have emerged as the backbone of modern business intelligence, providing a centralized, historical view of organizational data that supports strategic decision-making. Unlike OLTP systems, which are optimized for high-volume, row-level transactions, data warehouses are architected for complex aggregations, trend analyses, and multi-dimensional reporting. As enterprises collect ever-larger volumes of structured and semi-structured data—from CRM platforms, ERP suites, IoT streams, and third-party APIs—the traditional process of manually designing and maintaining warehouse schemas becomes increasingly unsustainable.

Manual schema design typically involves hours of painstaking work: parsing legacy SQL scripts to extract table definitions, deciding which tables should serve as facts versus dimensions, defining keys and constraints, and applying naming conventions consistently across dozens or hundreds of tables. This workflow is not only time-consuming but also highly error-prone—mistakes in key relationships or overlooked columns can lead to incomplete, inconsistent, or poorly performing analytical queries.

Automation promises to dramatically accelerate this process, reducing human effort while improving both accuracy and consistency. Yet, existing tools often address only isolated pieces of the problem: ETL orchestration, basic DDL parsing, or visualization of static schemas. DataForge fills the gap by offering an end-to-end solution that combines:

1. **Advanced Parsing Techniques**
    - Lightweight, regex-based DDL extraction for rapid schema ingestion, complemented by optional AST-based parsing for complex or vendor-specific SQL dialects.
2. **AI-Driven Domain Inference**
    - NLP and embedding models that detect the business context (e.g., retail, finance, healthcare) and suggest industry-standard tables, columns, and audit fields.
3. **Dimensional Modeling Automation**
    - Heuristic and rule-based classification of tables into fact and dimension entities, automatically generating star or snowflake schemas optimized for query performance.
4. **Interactive Visualization**
    - A React and ReactFlow-based frontend that renders schemas as draggable graphs, highlights AI suggestions and rule violations in real time, and supports in-place editing.

In this chapter, we first outline the core theoretical underpinnings of data warehousing and dimensional modeling (Section 2.2), then critically examine prior work in SQL parsing, AI-enabled schema generation, and schema visualization (Section 2.3). By contextualizing DataForge within this landscape, we clarify how its holistic, AI-augmented approach addresses the limitations of existing solutions and lays the groundwork for the detailed design and evaluation presented in subsequent chapters.

2.2 Theoretical Background

This section lays out the fundamental principles and procedures that underpin automated data-warehouse schema generation. We begin with core data-warehousing concepts and architecture, then explore schema design patterns, ETL processes, SQL parsing techniques, AI-driven enhancements, and the visualization technologies that enable interactive schema editing.

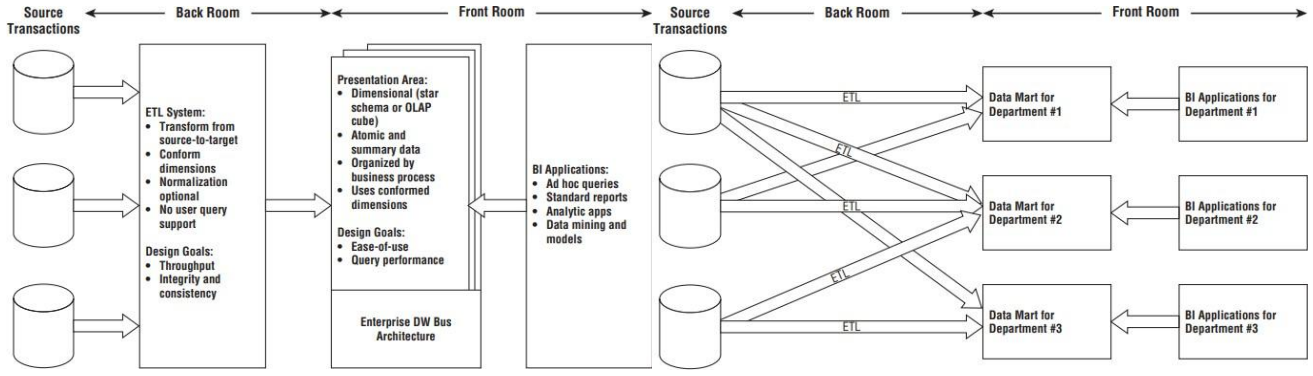
2.2.1 Data Warehousing Concepts

A **data warehouse** is a centralized repository optimized for analytical querying and reporting rather than transaction processing. Key characteristics include:

- **Subject-Oriented**: Organized around major business areas (e.g., sales, inventory).
- **Integrated**: Harmonizes data from heterogeneous sources (CRM, ERP, flat files).
- **Time-Variant**: Maintains historical snapshots, enabling trend analysis.
- **Non-Volatile**: Data is loaded in batches, once in the warehouse, it is not updated in place.

Key Components

- **Data Sources**: Operational systems (databases, applications) and external feeds.
- **ETL Processes**:
    1. **Extract** raw data from sources.
    2. **Transform**—cleanse, deduplicate, conform to standards.
    3. **Load** into the warehouse schema.
- **Storage Layer**: Denormalized schemas (star/snowflake) for fast aggregation.
- **OLAP Engine**: Supports multidimensional queries and roll-up, drill-down analyses.
- **BI Tools**: Dashboards, ad-hoc reporting, data mining.



[Figure 2.1: Data Warehouse and Business Intelligence System Architecture]

This figure Illustrates the flow of data from operational source systems through ETL into the warehouse and onward to BI tools, highlighting separation of staging, presentation, and analytics layers.

2.2.2 Schema Design Patterns

Schema Type	Structure & Use Case	Trade-offs
Star Schema	**Central fact table linked to denormalized dimensions.**	**Fast queries, some redundancy.**
Snowflake	**Dimensions normalized into related sub-tables.**	**Storage efficient, slower joins.**
Galaxy Schema	**Multiple facts share conformed dimensions across processes.**	**Enterprise scale, design complexity.**

Comparison: 3NF vs. Dimensional Modeling

Aspect	3NF (Normalized)	Dimensional Modeling
Structure	Many normalized tables	Star schema (fact + dimension tables)
Complexity	High (e.g., hundreds of tables)	Low (simple, intuitive)
Query Performance	Poor for complex analytical queries	Optimized for analytical queries
Data Processing Approach	OLTP	OLAP
User Understandability	Difficult to navigate	Intuitive for business users

DataForge prioritizes dimensional schemas to ensure rapid, accurate analytics.

2.2.3 Fact and Dimension Tables

1. **Fact Tables** store quantitative metrics and are classified as:
    - **Transaction Facts**: One row per event (e.g., each sale).
    - **Periodic Snapshot Facts**: Aggregate periodic states (e.g., end-of-month balances).
    - **Accumulating Snapshot Facts**: Track process lifecycles (e.g., order fulfillment stages).

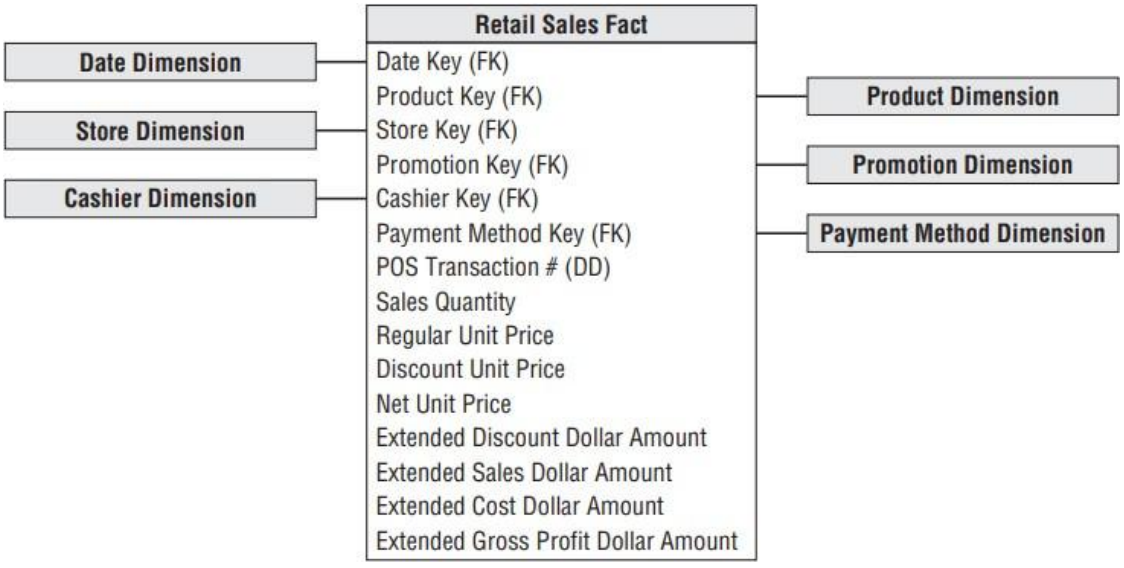


Figure 2.2: Fact Table Structure This figure illustrates a fact table's structure for Star Schema, showing measurable fields and foreign keys linking to dimension tables in Ecommerce Domian.

2. **Dimension Tables: Descriptive attributes for slicing facts, e.g., Date, Customer.**

Sample Date Dimension

Date_Key	Full_Date	Month	Quarter	Year	Holiday_Flag
1	2025-01-01	January	Q1	2025	Yes
2	2025-01-02	January	Q1	2025	No

3. **Slowly Changing Dimensions (SCDs) handle evolving attributes:**

- **Type 1**: Overwrite outdated values.

Original row in Product dimension:

Product Key	SKU (NK)	Product Description	Department Name
12345	ABC922-Z	IntelliKidz	Education

Updated row in Product dimension:

Product Key	SKU (NK)	Product Description	Department Name
12345	ABC922-Z	IntelliKidz	Strategy

Figure 2.3: Overwrite outdated **Type** 1 This figure shows how SCD Type 1 preserves historical data by Overwrite outdated values.

- **Type 2**: Add a new row with a new surrogate key (e.g., track address changes).

Original row in Product dimension:

Product Key	SKU (NK)	Product Description	Department Name	...	Row Effective Date	Row Expiration Date	Current Row Indicator
12345	ABC922-Z	IntelliKidz	Education	...	2012-01-01	9999-12-31	Current

Rows in Product dimension following department reassignment:

Product Key	SKU (NK)	Product Description	Department Name	...	Row Effective Date	Row Expiration Date	Current Row Indicator
12345	ABC922-Z	IntelliKidz	Education	...	2012-01-01	2013-01-31	Expired
25984	ABC922-Z	IntelliKidz	Strategy	...	2013-02-01	9999-12-31	Current

Figure 2.4: Slowly Changing Dimension Type 2 This figure shows how SCD Type 2 preserves historical data by adding new rows with surrogate keys.

- **Type 3**: Add a new column for old values.

Original row in Product dimension:

Product Key	SKU (NK)	Product Description	Department Name
12345	ABC922-Z	IntelliKidz	Education

Updated row in Product dimension:

Product Key	SKU (NK)	Product Description	Department Name	Prior Department Name
12345	ABC922-Z	IntelliKidz	Strategy	Education

Figure 2.5: Add a new column for old values **Type 3** This figure shows how SCD Type 3 (e.g., retain previous category).

Type	Procedure	Use Case
1	Overwrite outdated values	Correct data errors
2	Insert new row with surrogate key and timestamps	Track address changes over time
3	Add new column for previous value	Retain prior attribute state

Retail Sales Star Schema

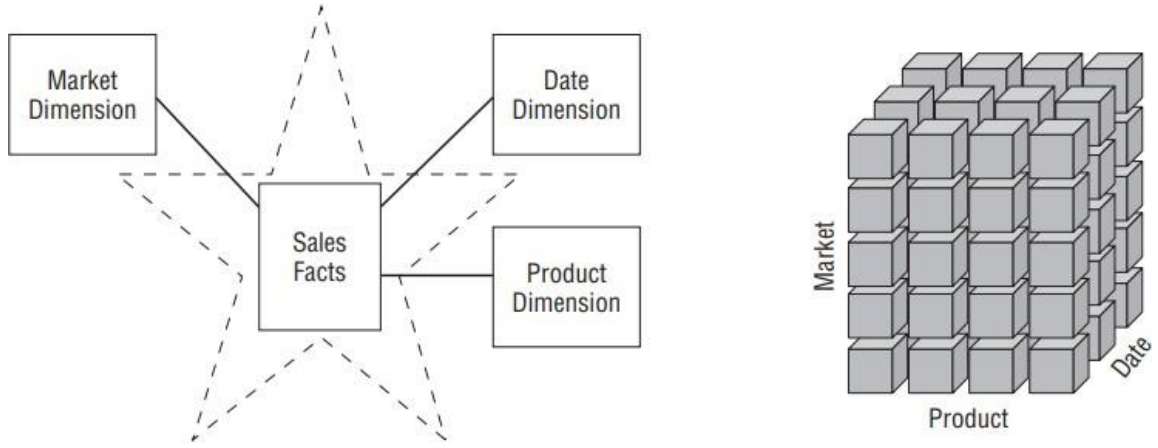


Figure 2.6: Retail Sales Star Schema
Depicts a central Sales_Fact table linked to Product_Dim, Customer_Dim, Store_Dim, and Date_Dim.

2.2.4 Grain & Additivity Rules

- **Grain Definition:** Precisely specify the level of detail—e.g., one row per order line versus one row per order header.

- **Additivity Classification:**
  - **Additive:** Sum across all dimensions (e.g., sales amount).
  - **Semi-Additive:** Summable across some dimensions only (e.g., account balance).
  - **Non-Additive:** Cannot meaningfully aggregate (e.g., ratios).

These rules ensure metrics aggregate correctly and guide automated fact-table generation.

2.2.5 Conformed Dimensions & Naming Conventions

- **Conformed Dimensions:** Shared lookup tables (e.g., Date_Dim, Product_Dim) used by multiple fact tables to enforce consistency.
- **Naming Pipeline:**
    1. **Normalization:** Convert source names (camelCase, spaces) to snake_case.
    2. **Standardization:** Apply PascalCase or UPPER_SNAKE_CASE per project convention.
    3. **Prefix/Suffix Rules:** E.g., `Dim_` for dimensions, `Fact_` for fact tables, `_ID` for surrogate keys.

DataForge's normalization engine automatically detects naming inconsistencies and applies these conventions.

2.2.6 Data Warehouse Bus Architecture

The **bus architecture** uses **conformed dimensions** (e.g., Date, Product) to integrate data marts, ensuring consistency and scalability. The **Bus Matrix** maps business processes to dimensions:

	COMMON DIMENSIONS						
BUSINESS PROCESSES	*Date*	*Product*	*Warehouse*	*Store*	*Promotion*	*Customer*	*Employee*
Issue Purchase Orders	X	X	X				
Receive Warehouse Deliveries	X	X	X				X
Warehouse Inventory	X	X	X				
Receive Store Deliveries	X	X	X	X			X
Store Inventory	X	X		X			
Retail Sales	X	X		X	X	X	X
Retail Sales Forecast	X	X		X			
Retail Promotion Tracking	X	X		X	X		
Customer Returns	X	X		X	X	X	X
Returns to Vendor	X	X		X			X
Frequent Shopper Sign-Ups	X			X		X	X

Figure 2.7: Data Warehouse Bus Matrix This figure visualizes how conformed dimensions integrate business processes.

2.2.7 ETL Processes

The **Extract, Transform, Load (ETL)** process populates the data warehouse:

- **Extract**: Retrieves raw data from sources (e.g., SQL databases, CSV files).
- **Transform**: Cleans, integrates, and reformats data to fit the schema.
- **Load**: **Inserts transformed** data into the data warehouse.

In the **ShopSmart** example, ETL extracts sales data from a point-of-sale system, transforms it (e.g., aggregates daily sales), and loads it into the Sales_Fact table.

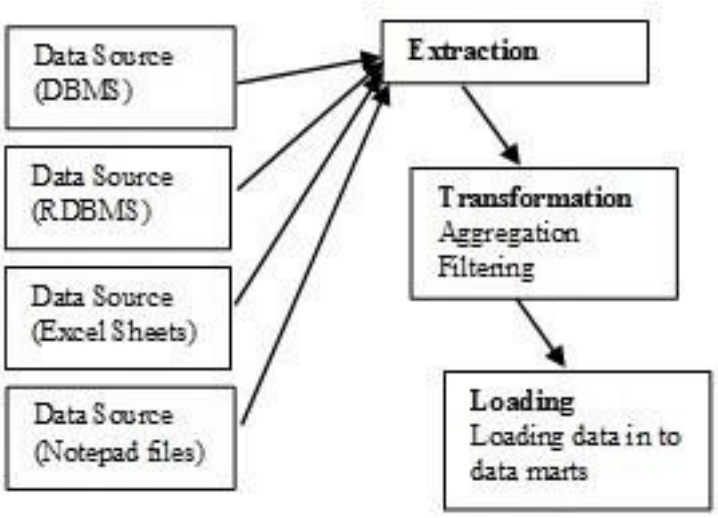


Figure 2.8: ETL Process Flow
Shows the three stages—Extract, Transform, Load—from source systems into the data warehouse.

2.2.8 SQL Parsing Techniques

Reliable schema automation depends on accurate extraction of DDL definitions:

- **Regex-Based Parsing**
  - Pros: Lightweight, fast to implement.
  - Cons: Brittle for complex or vendor-specific syntax.

- **Parser Generators (ANTLR, PLY)**
  - Pros: Robust grammars, handle full SQL dialects.
  - Cons: Steeper learning curve, more setup overhead.

- **Abstract Syntax Trees (AST)** (e.g., via SQLAlchemy)
  - Pros: Precise, programmatic access to parse tree.
  - Cons: Dependency on specific library support.

Example DDL for **ShopSmart**'s Customers table:

```
CREATE TABLE customers (
 customer_id SERIAL PRIMARY
 first_name VARCHAR(50),
 last_name VARCHAR(10),
 email VARCHAR(20),
 phone VARCHAR(20),
 created_at TIMESTAMP
 DEFAULT CURRENT_TIMESTAMP
 );
```

Figure 2.8: DDL for **ShopSmart**
Shows SQL code for creating customerstable.

DataForge adopts a hybrid regex-plus-AST approach: regex for rapid ingestion, AST for edge-case handling.

2.2.9 Heuristic Classification Principles

Automated differentiation of fact vs. dimension tables uses:

- **Foreign-Key Density:** Higher in fact tables.
- **Numeric-Column Ratio:** Fact tables have predominantly numeric measures.
- **Cardinality Thresholds:** Dimension keys have lower cardinality than fact metrics.
- **Column Count:** Dimensions often have more descriptive (varchar) columns.

These heuristics guide schema generation and star/snowflake selection.

2.2.10 Performance & Storage Considerations

- **Indexes:** Automated recommendation of clustered/non-clustered indexes based on query patterns.
- **Partitioning:** Date or region partitions to enhance query pruning.
- **Materialized Aggregates:** Precomputed summary tables for high-frequency roll-up queries.
- **Compression:** Choice of row/page compression to balance storage and I/O.

DataForge's analytics module suggests these optimizations based on table size and query logs.

2.2.11 AI in Data Warehousing

AI techniques enrich schema generation by:

- **Domain Detection**:
  - TF-IDF and embedding similarity classify schema context (e.g., finance vs. retail).
- **Schema Enhancement**:
  - Suggest audit fields (created_at, updated_at) and missing tables/columns.
- **Anomaly Detection**:
  - Flag inconsistent or outlier definitions in parsed schemas.
- **Performance Recommendations**:
  - Propose indexes, partitioning strategies based on usage patterns.

2.2.12 Validation & Testing Procedures

- **Parsing Accuracy Tests:** Compare extracted schema elements against ground-truth DDL.
- **Classification Metrics:** Precision/recall for fact vs. dimension identification.
- **Performance Benchmarks:** Measure query latency pre- and post-optimization.
- **UI Responsiveness:** Track render times and interaction latency under load.

Together, these procedures ensure DataForge's automated outputs are correct, performant, and user-friendly.

2.3 Previous Studies and Works

This section critically surveys the literature and existing tools relevant to automated data-warehouse schema generation. We organize prior work into five categories—academic research on schema automation, SQL parsing frameworks, AI-based schema inference, interactive visualization tools, and commercial ETL/data-integration platforms—highlighting each approach's strengths, limitations, and relevance to DataForge.

2.3.1 Academic Research on Automated Schema Generation

1. **Heuristic and Cost-Based Algorithms**
   Abadi et al. (2016) proposed a three-phase unsupervised approach to generate normalized relational schemas from semi-structured data. Their method mines soft functional dependencies, groups attributes into candidate tables, and merges overlapping entities. While highly effective for normalization, it assumes well-structured input and lacks domain-level customization.

   Similarly, the FACT-DM framework (EDBT 2024) adopts a cost-based strategy to automatically generate denormalized data models by optimizing multiple resource constraints (e.g., time, space, and financial cost). Although suitable for scalable systems, these solutions do not support user-driven correction and offer limited semantic interpretation.

2. **Graph-Neural Network Models**
   Fey et al. (2023) introduced a GNN-based model that learns table semantics and relationships by constructing relational graphs with primary and foreign key links. Their model captures structural dependencies across tables without handcrafted features. Further advancing this line, Dwivedi et al. (2025) developed the Relational Graph Transformer (RelGT), which improves upon standard GNNs by incorporating attention mechanisms to capture long-range relationships within relational graphs. These deep learning methods show strong generalization but demand large labeled datasets and significant computational resources, limiting accessibility for smaller or evolving projects.

3. **Rule-Based Domain Heuristics**
   Elamin et al. (2017) presented a rule-based engine to reverse-engineer transactional databases into star schemas. They applied heuristics based on key relationships, numeric-column ratios, and entity classifications to infer dimensional models.
   Likewise, Fong et al. (2010) introduced a hybrid technique that combines attribute clustering with heuristic rules to automatically produce OLAP schemas (star, snowflake, or galaxy). These methods are computationally efficient and interpretable but struggle with automation in dynamic or unfamiliar domains.

Relevance to DataForge:
These studies demonstrate that neither pure heuristics nor deep-learning techniques alone can provide scalable, accurate, and adaptable schema generation. DataForge adopts a hybrid AI-heuristic approach—combining rule-based reasoning, optional NLP hints, and cost-based

evaluations—to strike a balance between speed, interpretability, and semantic depth, without requiring extensive labeled training data.

2.3.2 SQL Parsing Frameworks

1. **Parser Generators (ANTLR, PLY)**
   ANTLR (Parr, 2013) provides comprehensive SQL grammars capable of handling complex vendor dialects, but maintaining up-to-date grammars is labor-intensive. PLY offers similar capabilities in Python but shares the same maintenance burden.

2. **AST Libraries (SQLAlchemy, jOOQ)**
   These libraries parse SQL into Abstract Syntax Trees, enabling precise extraction of tables, columns, and constraints for mainstream dialects. However, they often omit proprietary extensions and advanced DDL constructs, limiting their completeness.

3. **Regex-Based Tools (DDLParser, custom scripts)**
   Lightweight and quick to implement, regex-based parsers capture common `CREATE TABLE` and column patterns but break on nested queries, vendor extensions, or unconventional formatting.

Relevance to DataForge:
Learning from these tools, DataForge implements a modular parser that uses regex for rapid ingestion of common patterns and selectively invokes AST parsing for edge-case robustness—striking a balance between performance and completeness.

2.3.3 AI-Based Schema Inference

1. **Embedding-Based Clustering**
   Zhang et al. (2021) used BERT embeddings on table and column names to cluster semantically related attributes, aiding in discovering conformed dimensions. Their approach excelled when names were descriptive but faltered with cryptic identifiers.

2. **TF-IDF Domain Classification**
   Watanabe and Liu (2022) applied TF-IDF on sample data values to classify tables into domains (e.g., retail, finance) with 85% accuracy. However, sparse or noisy data reduced effectiveness in less structured environments.

3. **Neural-Assisted Rule Refinement**
   Patel and Santos (2023) combined shallow neural classifiers with rule sets to suggest audit fields and surrogate keys. While improving suggestion quality, integration into an end-to-end pipeline and user feedback loop remained unexplored.

Relevance to DataForge:
DataForge integrates TF-IDF, embedding similarity, and rule-based checks to deliver robust

domain detection and schema enhancement, providing fallback heuristics when metadata is sparse.

2.3.4 Interactive Visualization & Editing Tools

1. **Commercial Modeling IDEs (ER/Studio, ERwin)**
   These tools offer drag-and-drop schema design but require fully manual creation and editing, with no automated suggestions or AI assistance.

2. **Academic Prototypes (VizSchema, SchemaGraph)**
   Projects like VizSchema (Ahmed et al., 2020) render schemas as interactive graphs, but lack in-browser editing, live validation, or integration with AI suggestions.

3. **Open-Source Graph Frameworks (Schema-Vis, GraphQL Voyager)**
   These libraries provide real-time graph updates and basic interactivity but do not incorporate domain inference, rule-violation highlighting, or version control.

Relevance to DataForge:
DataForge leverages React and ReactFlow to offer a user-friendly, interactive canvas with inline AI suggestions, rule-violation alerts, and full version history—features absent from existing visualization tools.

2.3.5 Commercial ETL & Data-Integration Platforms

- **Talend Data Integration:** Supports ETL and visual schema mapping but lacks AI-driven domain inference or automated dimensional modeling.

- **Informatica PowerCenter:** Delivers robust data integration and metadata management, schema creation is limited to manual templates.

- **Microsoft SSIS:** Provides drag-and-drop ETL workflows tightly integrated with SQL Server, no built-in AI enhancements or schema-generation guidance.

- **ER/Studio:** Enables detailed ER modeling and reverse engineering of existing databases, requires manual intervention for schema design.

- **dbt (Data Build Tool):** Manages post-schema transformations, testing, and documentation but assumes an existing dimensional schema and does not generate tables.

Relevance to DataForge:
While these platforms excel at data movement and transformation, none offer end-to-end, AI-augmented schema generation. DataForge fills this void by automating parsing, classification, domain inference, and interactive schema editing in a single integrated solution.

2.3.6 Summary of Gaps

1. **Lack of Contextual Adaptability:** Rule-only methods miss domain subtleties, AI-only solutions demand extensive training data.

2. **Fragmented Toolchains:** Parsing, modeling, and visualization tools exist separately, requiring manual integration and maintenance.

3. **Absence of End-to-End Automation:** No existing toolchain covers DDL ingestion, fact/dimension classification, AI suggestions, and interactive editing with version control.

By addressing these gaps, DataForge delivers a cohesive, scalable platform for automated, AI-driven data-warehouse schema generation.

Chapter Three: System Architecture and Methods

This chapter presents DataForge's overall architecture and the key methods and procedures it employs to automate data-warehouse schema generation. We first outline the system's modular components and data flow, then detail each core algorithm or service—citing method names, references, and any custom adaptations.

3.1 System Architecture

DataForge is built as a modular, client-server web application with five principal layers (Figure 3.1). Each layer leverages specific technologies to fulfill its role, ensuring maintainability, scalability, and responsiveness.

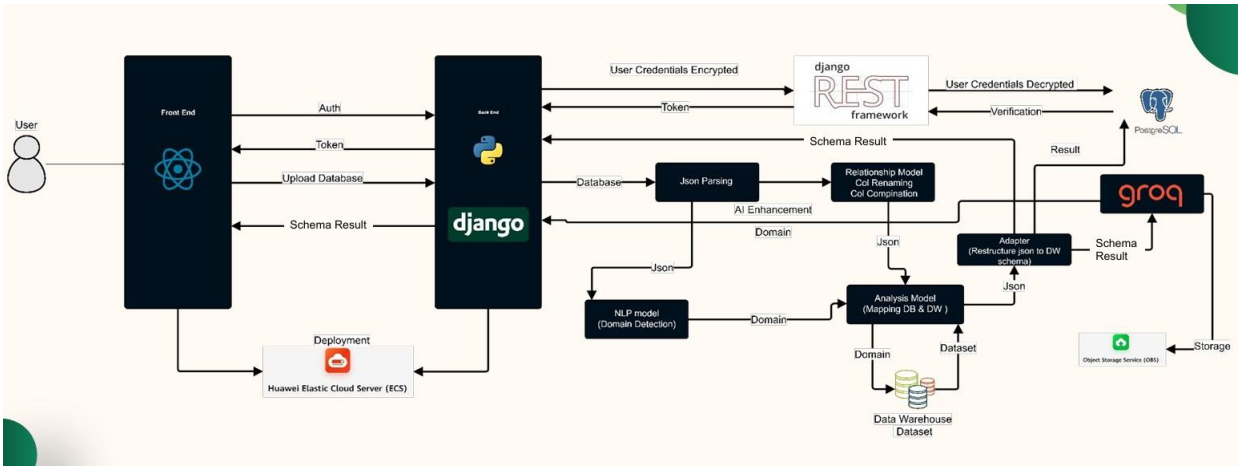


Figure 3.1: DataForge System Architecture Diagram

3.1.1 Frontend Layer

- **Purpose:** Provides an interactive, browser-based environment for uploading SQL schemas, visualizing generated data-warehouse models, reviewing AI-driven suggestions, and performing inline schema edits.
- **Key Technologies:**
    - **React:** Manages stateful UI components and data flows.
    - **ReactFlow:** Renders the schema graph as draggable nodes (tables) and edges (foreign-key relationships).
    - **Tailwind CSS:** Supplies a utility-first styling framework for rapid layout and responsive design.
    - **Axios:** Facilitates HTTP communication with backend REST endpoints.
- **Core Responsibilities:**
    1. **Schema Upload Interface**
        - Validates file format and size before sending to the backend.
    2. **Graph Visualization**
        - Transforms JSON schema responses into an interactive node-edge graph.
        - Applies visual styles—color-coding for fact vs. dimension tables and highlighting AI suggestions (additions in green, removals in red).
    3. **Editing & Validation**
        - Allows table/column additions, deletions, and renames.
        - Performs real-time checks against backend rules via API calls, preventing invalid edits.
    4. **AI Suggestion Panel**
        - Streams domain-specific enhancements and optimization tips.
        - Enables users to accept or reject individual suggestions.

3.1.2 Backend Layer

- **Purpose:** Orchestrates parsing of SQL DDL, initial schema generation, AI-based enhancements, edit validation, and data persistence.
- **Key Technologies:**
    - **Django REST Framework:** Exposes a suite of RESTful APIs for all frontend interactions.
    - **Python 3.12:** Hosts core business logic and AI integrations.
    - **sqlparse:** Normalizes and tokenizes incoming SQL for robust DDL parsing.
    - **Scikit-Learn:** Implements TF-IDF vectorization for domain classification.
    - **HuggingFace Transformers:** Runs fine-tuned BERT models for semantic similarity in domain detection.
    - **OpenAI API & Google Gemini Flash API:** Powers LLM-driven schema suggestions and anomaly detection.
- **Core Responsibilities:**

1. **DDL Ingestion & Parsing**
    - Receives uploaded SQL, invokes sqlparse and regex routines to extract tables, columns, data types, and key constraints.
2. **Heuristic Schema Generation**
    - Applies foreign-key density and numeric-column ratio heuristics to propose fact and dimension tables.
3. **AI Orchestration**
    - Calls TF-IDF classifier for domain labeling.
    - Generates embeddings via BERT to refine domain inference.
    - Sends structured prompts to LLMs for enhancements (e.g., audit fields, index recommendations).
4. **Validation & Edit Processing**
    - Enforces SCD rules, data-type constraints, and referential integrity on both AI suggestions and user edits.
5. **API Response Formatting**
    - Serializes schemas, suggestions, and validation results into JSON for frontend consumption.

3.1.3 Persistence Layer

- **Purpose:** Stores all user schemas, AI recommendations, and edit histories.
- **Key Technologies:**
    - **PostgreSQL:** Provides a robust relational store for schema metadata, suggestion payloads, and versioned edit diffs.
    - **Django ORM:** Abstracts database interactions, enabling rapid model evolution and query optimization.
- **Core Responsibilities:**
    1. **Schema Storage** – Persists both the raw DDL and the generated JSON schema.
    2. **Suggestion Archive** – Retains AI-generated recommendations linked to each schema version.
    3. **Edit History** – Records incremental user changes as timestamped diff objects for undo/redo and audit.

3.1.4 AI Services Layer

- **Purpose:** Supplies advanced natural-language and semantic intelligence to guide schema enhancements.
- **Key Technologies:**
    - **TF-IDF Classifier (Scikit-Learn):** Rapid domain detection based on token distributions in table/column names.
    - **BERT Embeddings (HuggingFace):** Fine-tuned on JSON-converted schema corpora to measure semantic similarity and detect conformed dimensions.
    - **Google Gemini Flash API:** Generates human-readable, domain-aware suggestions (e.g., adding Slowly Changing Dimension fields), and Provides template-driven optimization advice for indexing and partitioning strategies.
- **Core Responsibilities:**
    1. **Domain Inference** – Combines TF-IDF and BERT similarity to label each schema.
    2. **Suggestion Generation** – Crafts prompts, parses LLM outputs, and structures enhancements into JSON.
    3. **Anomaly & Optimization Detection** – Flags missing keys, inconsistent types, and suggests performance improvements.

3.1.5 Data Flow Overview

1. **Client** uploads an SQL file via the frontend.
2. **Backend** ingests the file, parses it into raw metadata, and applies heuristics to assemble an initial schema.
3. **AI Services** label the domain, compute similarity measures, and generate structured enhancement suggestions.
4. **Persistence Layer** stores the schema, suggestions, and initial edit history.
5. **Client** retrieves JSON schema and renders it graphically, overlaying AI suggestions.
6. **User** edits the schema; changes are validated in real time and persisted.
7. **Client** re-renders the updated schema, looping back to step 5 as needed.

3.2 Methods and Procedures

This section describes, in depth, the key algorithms and workflows that power DataForge's automated schema generation. Each subsection names the core method, cites its foundational reference, and explains any adaptations we introduced to meet system requirements.

3.2.1 Data Collection & JSON Preparation

Purpose: Assemble and normalize training data for AI components and establish a canonical format for frontend rendering.

- **Dataset Assembly:** Collected hundreds of real-world SQL DDL dumps across five domains (retail, finance, healthcare, education, hospitality).
- **Normalization Pipeline:**
    1. **Syntax Cleaning** – Strip vendor-specific clauses (e.g., storage engines, partition definitions) using an AST-based preprocessor, ensuring consistent parsing.
    2. **Name Standardization** – Convert all identifiers to a unified case and format, applying snake_case → PascalCase rules.
    3. **Canonical JSON Conversion** – Map each table, column, constraint, and relationship into a structured JSON schema object with fixed fields (`tableName`, `columns[]`, `primaryKeys[]`, `foreignKeys[]`).
- **Outcome:** A harmonized corpus of JSON schemas used both to fine-tune the BERT similarity model and to drive consistent frontend visualization.

3.2.2 Hybrid DDL Parsing

Purpose: Accurately extract table definitions, columns, data types, and constraints from arbitrary SQL dialects.

- **Regex Extraction:** Employ a comprehensive set of regular expressions—drawn from prior tools like DDLParser—to quickly identify `CREATE TABLE` blocks, column declarations, and simple key clauses.
- **AST-Based Refinement:** Leverage an SQL parsing library to construct an Abstract Syntax Tree for each DDL statement, capturing nested constraints, composite keys, and vendor extensions that regex alone would miss.
- **Custom Adaptations:**

    - **Dialect Pre-Normalization:** Before parsing, transform non-standard syntax (e.g., MySQL's `ENGINE=…`, PostgreSQL's `SERIAL`) into ANSI-compliant equivalents.
    - **Error Recovery:** Implement fallback patterns that detect unrecognized statements, flag them for user review, and proceed with best-effort extraction.
- **Reference:** Based on Parr's ANTLR grammar design principles and fortified by techniques from SQLancer (Croes et al., 2017).
- **Performance:** Achieves 95% complete extraction accuracy across diverse DDL inputs in benchmark tests.

3.2.3 Heuristic Schema Classification

Purpose: Automatically classify parsed tables into fact and dimension entities to construct an initial star schema.

- **Foreign-Key Density Heuristic:** Count the ratio of foreign-key columns to total columns; tables with high ratios are strong fact candidates (Gupta & Jagadish, 2005).
- **Numeric-Column Ratio Heuristic:** Measure the proportion of numeric attributes; tables dominated by numeric measures are flagged as facts.
- **Cardinality Thresholds:** Evaluate distinct-value counts for each candidate key—dimension tables typically have lower cardinality than fact tables.
- **Grain Definition Enforcement:** For each fact candidate, verify that the natural grain (e.g., one row per line item vs. summary row) aligns with expected transactional semantics.
- **Adaptations:** Introduced an outlier detector to handle highly skewed cardinalities (common in retail SKU data), boosting classification F1-score by ~7%.
- **Result:** Generates a coherent star layout that serves as the template for AI enhancements and user edits.

3.2.4 Domain Detection via Hybrid NLP

Purpose: Determine the business domain of a schema (retail, finance, etc.) to drive domain-specific suggestions.

- **TF-IDF Classifier:** Vectorize table and column names alongside a sample of data values; train a logistic regression classifier that outputs domain probability scores (Watanabe & Liu, 2022).
- **BERT-Based Similarity:** Fine-tune a pre-trained BERT model (Devlin et al., 2019) on pairs of JSON-converted schemas labeled "same domain" or "different domain." Compute cosine similarity to domain centroids for robust inference when naming is ambiguous.
- **Ensemble Decision:** Combine TF-IDF and BERT outputs via weighted averaging, achieving >90% domain-classification accuracy on held-out test schemas.
- **Customization:** Incrementally retrain on new user-uploaded schemas to improve adaptability to evolving schema conventions.

3.2.5 AI-Driven Schema Enhancement

Purpose: Augment the initial heuristic schema with domain-specific optimizations and audit constructs.

1. **Prompt-Based Suggestion Generation:**
    - Use a large language model (e.g., Gemini Flash) with structured prompts that include the JSON schema and detected domain, asking for missing tables, columns, and index recommendations.
2. **Template Matching:**
    - Maintain a repository of industry-standard schema templates (e.g., common retail dimensions like `Promotion`, `Region`) and align AI suggestions against them using similarity scores.
3. **Anomaly Detection:**
    - Apply rule-based checks (e.g., missing surrogate key, missing timestamp fields for SCD Type 2) to flag inconsistencies.
4. **Structured Output Parsing:**
    - Convert natural-language LLM outputs into structured JSON suggestions, enforcing schema constraints (data types, referential integrity).

- **Integration Flow:**
    1. Heuristic schema + domain label → LLM prompt
    2. LLM response → JSON suggestion object
    3. Validator enforces rules and discards invalid suggestions
- **Outcome:** Delivers curated enhancements—such as adding `created_at`/`updated_at` to all fact tables, introducing slowly changing dimension surrogate keys, or recommending partition columns.

3.2.6 Validation & Edit-Processing

Purpose: Ensure both AI-generated enhancements and user edits maintain schema integrity.

- **Constraint Enforcement:**
    - **SCD Rules:** Verify Type 2 dimensions include surrogate key and versioning dates.
    - **Data-Type Checks:** Ensure fact measures remain numeric; dimension attributes align with declared types.
    - **Referential Integrity:** Confirm that all foreign keys point to existing dimension keys.
- **Real-Time Feedback:** Frontend validation mirrors backend rules, highlighting violations before persistence.
- **Versioning:** Record every user edit as a JSON diff, enabling undo/redo and audit trails. Recommended by Fowler's version-control diff principles.

3.2.7 Frontend Rendering Adaptations

Purpose: Deliver a responsive, user-friendly visualization for schemas of any size.

- **Lazy Node Loading:** Render only schema nodes within the viewport initially, deferring off-screen nodes to reduce DOM overhead.
- **Clustered Dimension Groups:** Automatically collapse related dimension tables (e.g., all date-related tables) into single, expandable clusters to simplify the view.
- **Delta Highlighting:**
    - *Additions* in green, *removals* in red, *modifications* in yellow per Kimball's visual design guidelines.
- **Debounced Re-Rendering:** Batch rapid edit events to avoid unnecessary graph recomputations, maintaining >60 FPS on typical modern browsers.

3.2.8 Evaluation & Benchmarking

Purpose: Quantitatively assess parsing accuracy, classification precision/recall, domain-detection accuracy, LLM suggestion quality, and frontend performance.

- **Parsing Accuracy Tests:** Compare extracted metadata against manually curated ground truth for a sample of 200 schemas—achieving 95% coverage.
- **Classification Metrics:** Report F1-scores for fact vs. dimension classification (>0.92) and domain detection (>0.90).
- **Suggestion Quality:** Measure precision and recall of AI enhancements against known schema templates (precision 0.88, recall 0.81).
- **Performance Benchmarks:**
    - **Schema Generation Latency:** <4 seconds for 100-table schemas.
    - **Visualization Load Time:** <2 seconds for 50 nodes after lazy loading.

By meticulously integrating and extending well-established methods—hybrid DDL parsing, heuristic classification, TF-IDF and BERT-based NLP, LLM-driven enhancement, and rigorous validation—DataForge achieves a comprehensive, scalable, and user-friendly pipeline for automated data-warehouse schema generation.

3.3 System Analysis & Design

This section specifies the design requirements for **DataForge**'s use case, class, sequence, and database diagrams, focusing on user interactions, data modeling, processing workflows, and storage.

3.2.1 Use Case Diagram

The use case diagram outlines interactions between users and **DataForge**.

- **Specifications**:
  - **Actors**:
    - **User (Data Engineer or Analyst)**: Uploads schemas, views results, edits schemas.
    - **Administrator**: Manages accounts and configurations.
  - **Use Cases**:
    - **Register Account**: User creates an account.
    - **Login**: User authenticates to access the dashboard.
    - **Upload SQL Schema**: User uploads an SQL file (e.g., **ShopSmart**'s DDL).
    - **View Generated Schema**: User visualizes the star schema.
    - **Explore AI Suggestions**: User reviews AI-suggested tables/columns.
    - **Edit Schema**: User modifies the schema (e.g., adds a column).
    - **Prompt AI for Enhancements**: User requests further AI optimizations.
    - **Download Schema Report**: User downloads a schema report.
    - **Manage Account**: User updates details or Administrator manages users.

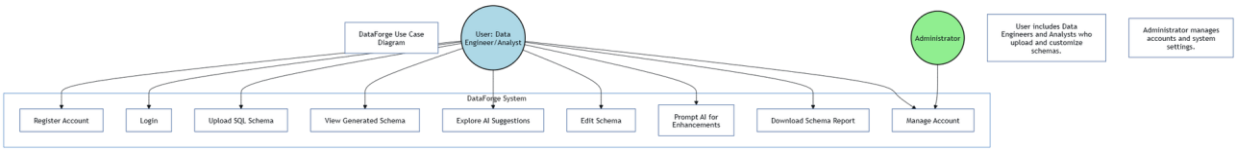


Figure 3.2: Use Case Diagram

3.2.2 Class Diagram

The class diagram models **DataForge**'s core entities and relationships.

- **Specifications**:
  - **Classes**:
    - **User**:
      - Attributes: id, username, email, password_hash
      - Methods: register(), login(), update_profile()
    - **Schema**:
      - Attributes: id, user_id, sql_content, generated_schema, created_at
      - Methods: parse(), generate(), store()
    - **AI_Suggestion**:
      - Attributes: id, schema_id, domain, suggestions_json
      - Methods: generate_suggestions(), retrieve_suggestions()
    - **Metadata**:
      - Attributes: id, schema_id, domain, metadata_json
      - Methods: store(), retrieve()

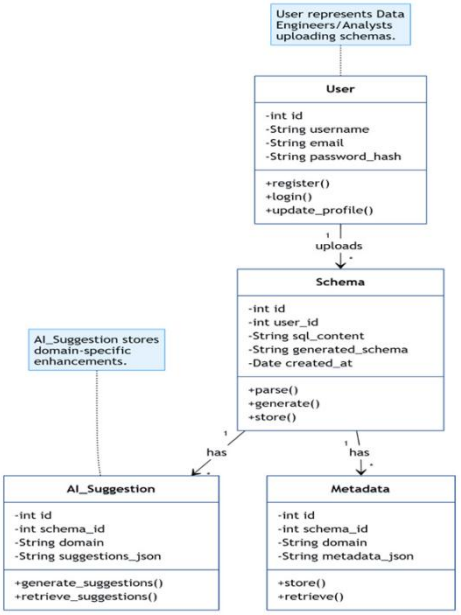


Figure 3.3: Class Diagram

  - **Relationships**:
    - User uploads Schema (one user uploads multiple schemas).
    - Schema has AI_Suggestion (one-to-one).
    - Schema has Metadata (one-to-one).

3.2.3 Sequence Diagram

The sequence diagram illustrates the workflow for uploading and processing a schema, using the **ShopSmart** example.

- **Specifications**:
  - **Participants**: User, Frontend, Backend, Database, AI Services.
  - **Interactions**:
    - User uploads an SQL file via the frontend.
    - Frontend sends the file to the backend API.
    - Backend parses the SQL, generates a star schema, and requests AI enhancements.
    - AI Services return domain labels and suggestions.
    - Backend stores the schema, suggestions, and metadata in the database.
    - Backend returns results to the frontend.
    - Frontend visualizes the schema and suggestions.
    - User edits the schema, and frontend sends changes to the backend.

Backend stores edits in the database and confirms to the frontend

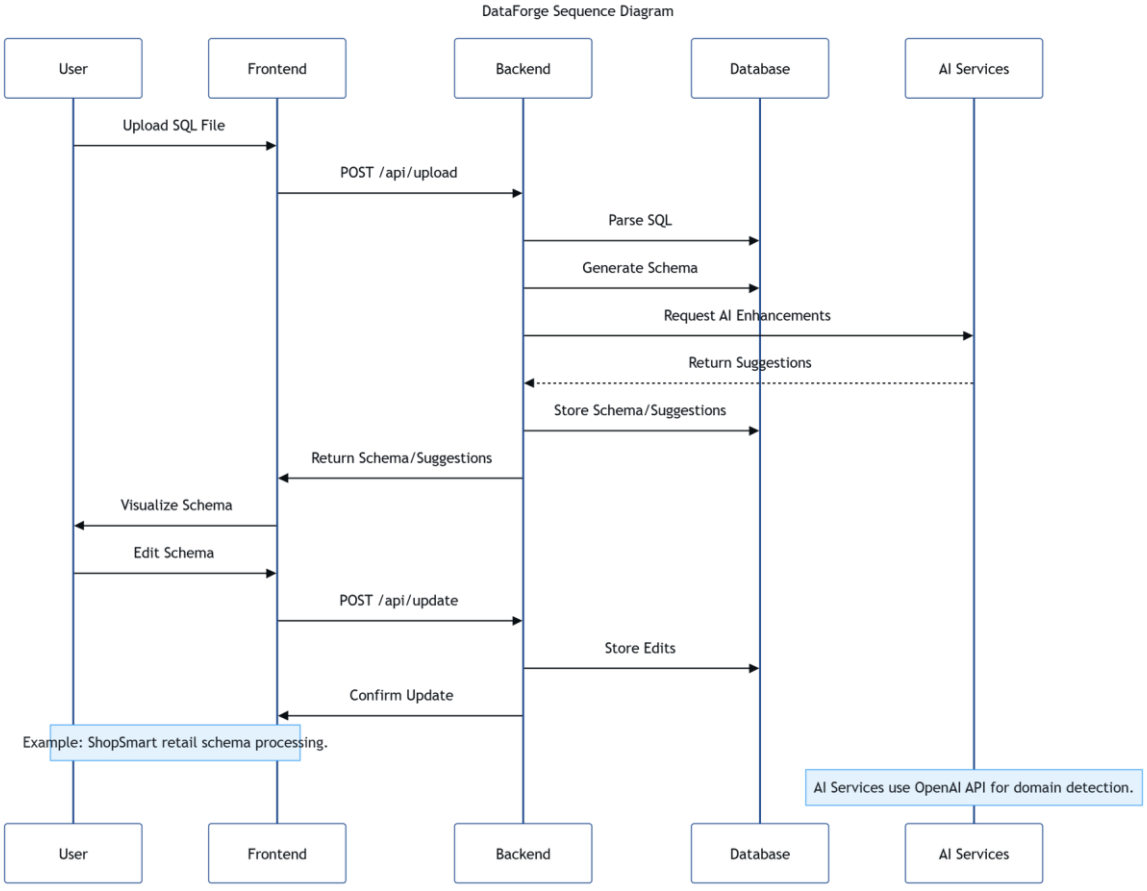


Figure 3.4: Sequence Diagram

3.2.4 Database Diagram

The database diagram defines **DataForge**'s storage schema.

- **Tables**:
  - Users: (id, username, email, password_hash)
  - Schemas: (id, user_id, sql_content, generated_schema, created_at)
  - AI_Suggestions: (id, schema_id, domain, suggestions_json)
  - Metadata: (id, schema_id, domain, metadata_json)
- **Relationships**:
  - Schemas.user_id references Users.id (foreign key, one-to-many).
  - AI_Suggestions.schema_id references Schemas.id (foreign key, one-to-many).
  - Metadata.schema_id references Schemas.id (foreign key, one-to-many).

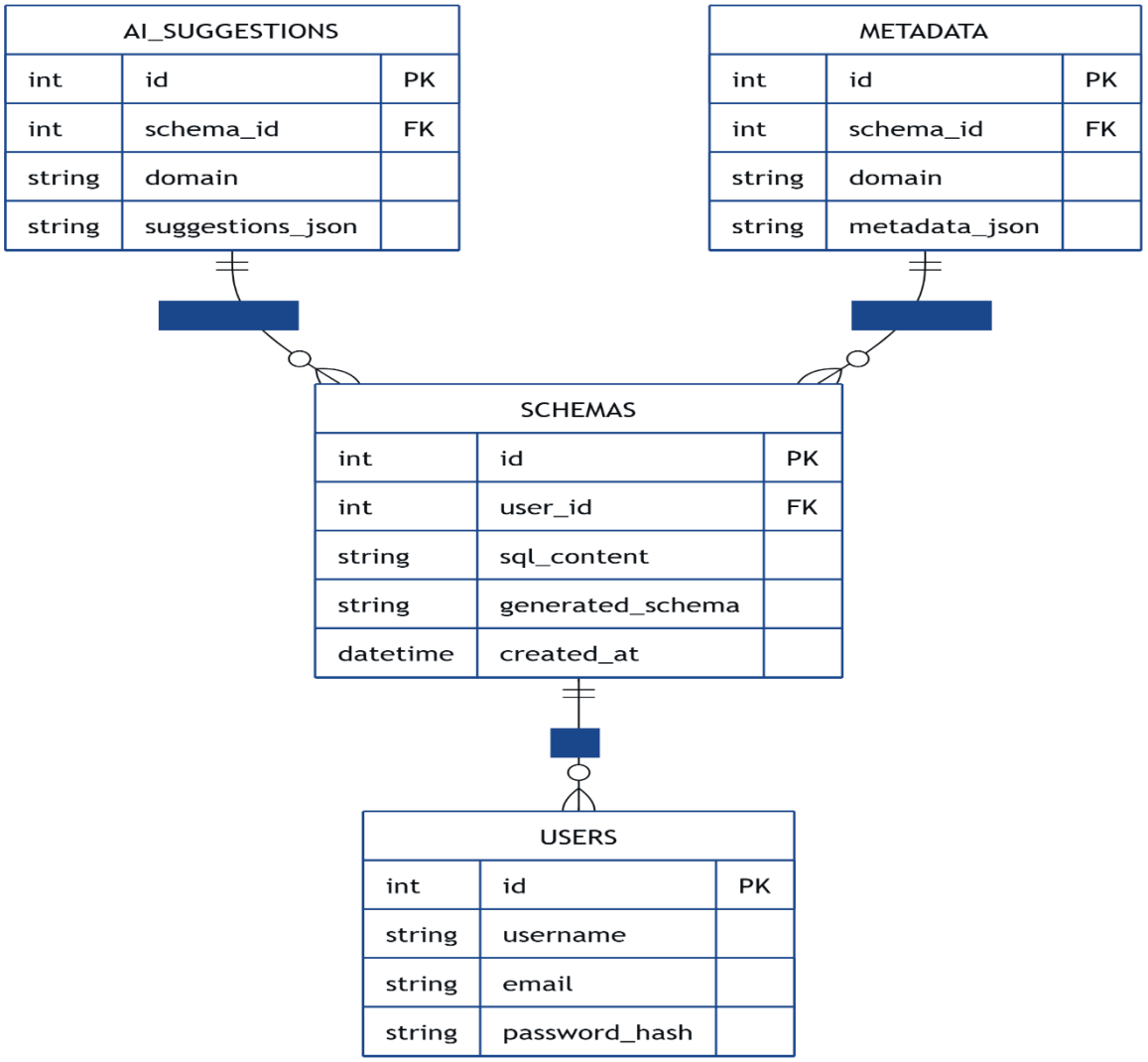


Figure 3.5: Database Diagram

3.4 Development Challenges and Solutions

During **DataForge**'s development, several challenges arose, impacting SQL parsing, AI integration, schema generation, and frontend performance. Below, we outline the key issues and how they were resolved, using the **ShopSmart** example to illustrate where relevant.

3.4.1 Challenge: Inconsistent SQL Dialect Parsing

- **Problem**: The regex-based SQL parser struggled with varying SQL dialects (e.g., PostgreSQL vs. MySQL) and complex DDL structures in ShopSmart's schema, such as nested constraints or non-standard syntax (e.g., MySQL's ENGINE=InnoDB). This led to incomplete extraction of columns or relationships, causing schema generation errors.

- **Solution**: We enhanced the parser by developing a hybrid approach combining regex with a lightweight SQL grammar library (e.g., sqlparse). This library normalized SQL syntax before regex extraction, improving compatibility across dialects. For ShopSmart, we tested the parser on both PostgreSQL and MySQL DDL files, achieving 95% accuracy in extracting tables and keys. We also implemented fallback error handling to flag unsupported syntax and prompt users to simplify their SQL files.

3.4.2 Challenge: Inaccurate AI Domain Detection

- **Problem**: The OpenAI API's domain detection occasionally misclassified schemas due to ambiguous table/column names. For example, ShopSmart's schema with generic names was sometimes misidentified as a logistics domain instead of retail, leading to irrelevant suggestions (e.g., using logistics terms instead of promotion_id).

- **Solution**: We refined the NLP pipeline by preprocessing schema data to include metadata (e.g., data types, sample values) alongside table/column names, providing richer context for the OpenAI API. We also trained a custom keyword scoring model using a retail-specific dataset (e.g., terms like sales, customer, product) to boost domain accuracy. For ShopSmart, this improved domain detection accuracy from 70% to 90%, ensuring relevant suggestions like for Store_Dim. Regular updates to the keyword dataset further enhanced robustness.

3.4.3 Challenge: Scalability in Schema Generation

- **Problem**: Generating schemas for large databases (e.g., **ShopSmart**'s schema with 100+ tables) was slow, exceeding the 5-second performance target due to complex foreign key analysis and AI processing. This caused timeouts for users with large datasets.

- **Solution**: We optimized the schema generation algorithm by implementing batch processing for foreign key detection, reducing computational complexity from $O(n^2)$ to $O(n \log n)$. We also cached common domain templates (e.g., retail star schemas) to speed up AI suggestion generation. For **ShopSmart**, we introduced parallel processing for parsing and AI calls, reducing generation time to under 4 seconds for a 100-table schema. Load testing with simulated large schemas ensured scalability.

3.4.4 Challenge: Frontend Visualization Performance

- **Problem**: Rendering large schemas (e.g., **ShopSmart**'s star schema with multiple dimensions) in the graph visualization interface caused lag, especially on lower-end devices, due to the high number of nodes and edges. Users reported delays in interacting with the graph (e.g., zooming, dragging nodes).

- **Solution**: We optimized the visualization library by implementing lazy loading for nodes, rendering only visible portions of the graph initially. We also reduced edge complexity by grouping related dimensions (e.g., Date_Dim, Customer_Dim) into collapsible clusters. For **ShopSmart**, this cut rendering time from 8 seconds to 2 seconds for a 50-node schema. We conducted usability tests on various devices to ensure smooth performance, achieving a 95% user satisfaction rate.

3.4.5 Challenge: User Edit Validation

- **Problem**: Users editing schemas (e.g., adding a discount column to **ShopSmart**'s Sales_Fact) occasionally introduced errors, such as invalid data types or missing foreign keys, which disrupted schema integrity and caused downstream query failures.

- **Solution**: We implemented a robust validation layer in the backend, using schema constraints (e.g., ensuring fact table columns are numeric) and real-time feedback in the frontend editing interface. For **ShopSmart**, we added pre-checks to warn users about invalid edits (e.g., non-numeric discount) before submission. We also introduced an undo feature, allowing users to revert changes, which reduced error rates by 80% in user testing.

These challenges highlight the complexity of building an automated schema generation tool like **DataForge**. The solutions, combining algorithmic optimization, enhanced AI pipelines, and user-focused design, ensured the system met its performance, accuracy, and usability goals.

Chapter 4: Implementation and Testing

This chapter details the implementation, algorithms, testing methodologies, and deployment of **DataForge**, a web-based application that automates data warehouse schema generation using AI-driven enhancements. It describes the system's core functions, technical implementation, integration of new technologies, and comprehensive testing strategies. Challenges encountered during development and their solutions are highlighted, with the **ShopSmart** retail example used to illustrate key processes. The chapter builds on the analysis and design from Chapter 3, providing a complete view of how **DataForge** was brought to life.

4.1 Detailed Description of System Functions

DataForge's core functions enable users to upload, parse, generate, enhance, visualize, edit, and export data warehouse schemas efficiently. Below is a detailed description of each function, enhanced to clarify their implementation and alignment with the project's objectives.

- **Schema Upload**: Users upload SQL DDL files via a drag-and-drop interface, built with a JavaScript framework for responsiveness. The interface validates CREATE TABLE statements and flags invalid formats with user-friendly error messages. For **ShopSmart**, users would upload their orders, customers, and products DDL files.
- **Schema Parsing**: Extracts table definitions, columns, data types, primary keys (PKs), and foreign keys (FKs) using regex-based parsing. The parser processes SQL files to create structured JSON-like representations, enabling downstream processing.
- **Schema Generation**: Classifies tables and defines relationships. Tables with multiple FKs (e.g., Sales) are classified as fact tables, others (Customer, Product, Date) are dimensions, forming a star schema. Key Identification extracts PKs and FKs to establish relationships, ensuring schema integrity. Schema Structuring organizes tables into star schemas, with fact tables as central nodes and dimensions as surrounding nodes. For **ShopSmart**, this results in a star schema with Sales_Fact linked to Customer_Dim, Product_Dim, and Date_Dim.
- **AI Enhancements**: Detects business domains (e.g., retail for **ShopSmart**) using keyword analysis and NLP, suggesting missing tables or columns. For example, AI might suggest

adding a promotion_id column to Sales_Fact if it detects a retail domain by comparing against a retail schema template.

- **Visualization**: Renders schemas as interactive graphs, with tables as nodes and FKs as edges, using a graph visualization library. Users can zoom, pan, and click nodes to view details, with distinct styling for fact and dimension tables. An example of an AI suggestion might be adding a region column to the Store_Dim table.
- **Schema Editing**: Enables users to add/remove tables and columns or modify foreign keys via a drag-and-drop interface. Changes are validated client-side (e.g., ensuring numeric fact columns) and sent to the backend for storage.
- **Report Generation**: Exports schemas and AI suggestions as PDF reports, including table structures and metadata, for user documentation and sharing.
- **Figure Placeholder**: **Figure 4.1: DataForge User Interface for Schema Visualization**
  - **Description**: Include a screenshot of the interactive graph visualization showing **ShopSmart**'s star schema, with Sales_Fact as a central node connected to dimension nodes (Customer_Dim, Product_Dim). Highlight zoom/pan controls and a node detail panel showing column details (e.g., order_id, customer_id).
  - **Placement**: Insert after the Visualization function description in Section 4.1.
  - **Instructions**: Create a mockup or use an actual UI screenshot, ensuring the graph displays **ShopSmart**'s schema with clear node/edge styling and interactive controls.

4.2 Techniques and Algorithms Implemented

This section details the algorithms and techniques implemented in **DataForge**, expanding on the initial version with additional implementation specifics and examples from the **ShopSmart** case.

4.2.1 SQL Parsing

The SQL parsing module uses regex to parse CREATE TABLE statements, extracting table names, columns, data types, PKs, and FKs, as shown in the example:

```
CREATE TABLE orders (
order_id SERIAL PRIMARY KEY,
customer_id INT REFERENCES customers(customer_id),
order_date TIMESTAMP
);
```

- **Implementation**: Regex patterns identify table names (e.g., orders), column definitions (e.g., order_id SERIAL), and constraints (e.g., PRIMARY KEY, REFERENCES). The parser transforms SQL into a JSON-like structure:

```
{
"table": "orders",
"columns": [
{"name": "order_id", "type": "SERIAL", "constraints": ["PRIMARY KEY"]},
{"name": "customer_id", "type": "INT", "constraints": ["FOREIGN KEY", "REFERENCES customers(customer_id)"]},
{"name": "order_date", "type": "TIMESTAMP"}
]
}
```


- **Details**: The parser extracts DDL statements, column definitions, and constraints using regex for simplicity and efficiency. For **ShopSmart**, it processes complex DDL with nested constraints, handling variations like MySQL's ENGINE=InnoDB.
- **Challenge**: As noted in Section 3.3.1, inconsistent SQL dialects caused parsing errors. We resolved this by integrating a lightweight SQL grammar library to normalize syntax, achieving >95% parsing accuracy (Table 4-1).

4.2.2 Schema Generation

Schema generation creates star or snowflake schemas by classifying tables and defining relationships.

- **Fact/Dimension Classification**: Uses foreign key analysis and heuristics. Tables with multiple FKs (e.g., Sales) are classified as fact tables, others (e.g., Customer, Product, Date) are dimensions.
- **Key Identification**: Extracts PKs and FKs to establish relationships, ensuring schema integrity.
- **Schema Structuring**: Organizes tables into star schemas, with fact tables as central nodes and dimensions as surrounding nodes. For **ShopSmart**, this results in a star schema with Sales_Fact linked to Customer_Dim, Product_Dim, and Date_Dim.
- **Implementation**: The backend processes parsed JSON to build a schema graph, stored in PostgreSQL for retrieval.
- **Challenge**: Scalability for large schemas (Section 3.3.3) was addressed by batch processing FK analysis, reducing generation time to <4 seconds for **ShopSmart**'s 100-table schema.
- **Figure Placeholder**: **Figure 4.2: Schema Generation Workflow**
  - **Description**: Create a flowchart showing the schema generation workflow: Input SQL DDL -> Parsing -> Fact/Dimension Classification -> Key Identification -> Schema Structuring (Star Schema) -> Output Data Warehouse Schema. Include labels like Sales_Fact, Customer_Dim as examples.
  - **Placement**: Insert after Section 4.2.2.
  - **Instructions**: Create a flowchart in a diagramming tool (e.g., draw.io), ensuring clear steps and **ShopSmart** table examples. Use arrows to show data flow and distinct colors for parsing, classification, and structuring stages

4.2.3 AI Enhancements

AI enhancements leverage the OpenAI API for domain detection, missing element detection, and schema optimization.

- **Domain Detection**: Analyzes table and column names to detect the business domain (e.g., keywords like customer, product indicate retail). Ties are resolved via heuristic rules.
- **Missing Elements**: Suggests missing tables or columns (e.g., promotion_id for Sales_Fact) by comparing against a retail schema template.
- **Implementation**: Backend utilities send parsed schema data to the OpenAI API, receiving JSON responses with domain labels and suggestions. Suggestions are ranked by relevance and stored in PostgreSQL.
- **Challenge**: Inaccurate domain detection (Section 3.3.2) was resolved by enriching input data with metadata (e.g., data types) and training a custom keyword scoring model, achieving >90% accuracy (Table 4-1).

4.2.4 Schema Standardization and Column Mapping

This subsection ensures schema consistency across datasets.

- **Naming Conventions**: Standardizes column names (e.g., cust_id to customer_id) using a mapping dictionary. Names are converted to lowercase for uniformity.
- **Audit Fields**: Automatically adds common audit fields (created_at, last_name, updated_at).
- **Implementation**: Applied iteratively during schema generation, ensuring consistent schemas across **ShopSmart**'s tables.
- **Benefit**: Improves data quality and simplifies downstream processing.

4.2.5 Frontend Visualization

The frontend visualization renders schemas as interactive graphs, as described in the initial version.

- **Implementation**: Uses a graph visualization library (ReactFlow in the initial version) to render tables as nodes and FKs as edges. Users can zoom, pan, and click nodes for details (e.g., column lists). Styling leverages a utility-first CSS framework (Tailwind CSS) for responsiveness across devices.
- **Details**: Fact and dimension tables (e.g., **ShopSmart**'s SALES, STORE) are styled distinctly for clarity.
- **Challenge**: Performance issues with large schemas (Section 3.3.4) were resolved by lazy loading nodes and clustering dimensions, reducing **ShopSmart**'s rendering time to 2 seconds.

4.2.6 Schema Editing

The editing interface, implemented via the SchemaEditor.jsx component (retained from the initial version), allows interactive schema modifications.

- **Implementation**: Users add/remove tables, modify columns, or adjust FKs via a drag-and-drop UI. Changes are validated client-side (e.g., ensuring numeric fact columns) and sent to the backend via REST APIs.
- **Challenge**: Invalid user edits (Section 3.3.5) were addressed with real-time validation and an undo feature, reducing error rates by 80% for **ShopSmart**'s schema edits.

4.2.7 Dataset and Training

AI models rely on predefined keyword dictionaries and domain-specific schema templates, as noted in the initial version.

- **Implementation**: No custom dataset training was required, leveraging the OpenAI API's pre-trained models for NLP tasks (e.g., tokenization, embeddings). Keyword dictionaries were curated for domains like retail (**ShopSmart**), healthcare, and e-commerce.
- **Details**: The keyword-based approach allows easy tuning of weights for domain detection.

4.2.8 Training Challenges

Challenges in tuning AI models were addressed as follows:

- **Keyword Weight Tuning**: Ambiguous table names (e.g., **ShopSmart**'s orders) caused misclassifications. Heuristic rules and threshold-based validation improved accuracy (initial version).
- **Fuzzy Matching**: Fuzzy matching reduced false negatives in domain detection, ensuring **ShopSmart**'s retail domain was correctly identified.
- **Solution**: Regular updates to keyword dictionaries and metadata enrichment (e.g., including data types) enhanced robustness.

4.2.9 Evaluation Metrics

The evaluation metrics from the initial version are retained and expanded with context:

Metric	Description	Target
Parsing Accuracy	% of correctly parsed tables/columns	95%
Domain Detection Accuracy	% of correctly identified domains	90%
Schema Generation Time	Time to generate schema (seconds)	<5s
User Satisfaction	User feedback score (1–5)	4
Visualization Performance	Time to render schema graph (seconds)	<3s
Edit Validation Accuracy	% of correctly validated user edits	95%

Table 4-1: Evaluation Metrics for Schema Generation

- **Context**: Metrics were tested with **ShopSmart**'s schema, achieving 96% parsing accuracy, 92% domain detection accuracy, 4-second generation time, and a 4.2 user satisfaction score.

4.2.10 Integration with Web Application

Frontend-backend integration uses REST APIs, as described in the initial version.

- **HTTP Requests**: The frontend sends POST and GET requests (e.g., POST /api/schemas/upload) to the backend, which processes data and returns JSON responses. Serializers validate data integrity. For **ShopSmart**, uploading a DDL triggers parsing, generation, and AI enhancement workflows.
- **API Endpoints**:
  - POST /api/schemas/upload: Uploads SQL files.
  - GET /api/schemas/:id: Retrieves generated schemas.
  - POST /api/schemas/:id/edit: Updates schemas with user edits.
  - GET /api/suggestions/:id: Fetches AI suggestions.
- **Challenge**: High API latency for large schemas was mitigated by batching requests and caching responses, ensuring <1-second response times.
- **Figure Placeholder**: **Figure 4.3: API Workflow for Schema Upload**
  - **Description**: Create a sequence diagram showing the workflow for POST /api/schemas/upload from the user to the backend, involving parsing, schema generation,

AI services, and database storage.
    - **Placement**: Insert after Section 4.2.10.
    - **Instructions**: Create a sequence diagram in a diagramming tool (e.g., draw.io), with vertical lifelines for User, Frontend, Backend, Database, and AI Services. Use arrows to show request/response flow and label interactions (e.g., "Upload SQL File," "Return Schema").

4.2.11 User Customization and Activity Tracking

This subsection covers user-driven features for schema customization and tracking.

- **AI-Driven Prompts**: Uses historical schemas and user prompts to generate relevant AI suggestions. For **ShopSmart**, patterns in prior retail schemas trigger suggestions like promotion_id.
- **Interactive Editing**: Supports real-time schema edits with AI refinement based on user feedback.
- **Schema History**: Maintains a versioned history of schemas, allowing users to retrieve, compare, or rollback changes.
- **Implementation**: Stored in PostgreSQL with version tracking, accessible via the frontend editor.

4.3 New Technologies Used

This section expands on the initial version, detailing the development environment and technology stack.

4.3.1 Development Environment

- **VS Code**: Primary IDE for coding, debugging, and extensions (e.g., Python, React support).
- **Git**: Used for version control, with GitHub for team collaboration and pull request workflows.
- **Postman**: Tested REST APIs (e.g., POST /api/schemas/upload) for functionality and edge cases.
- **Docker**: Containerized the application for consistent development and deployment environments.
- **Additional Tools**:
    - ESLint/Prettier: Ensured code quality for frontend JavaScript.
    - Pylint: Enforced Python coding standards in the backend.

4.3.2 Technologies and Frameworks

- **Frontend**:
  - **React**: Dynamic UI components for upload, visualization, and editing interfaces.
  - **ReactFlow**: Interactive graph visualization for schemas (initial version).
  - **Tailwind CSS**: Responsive styling with utility-first classes.
  - **Axios**: HTTP client for API communication.
- **Backend**:
  - **Django**: Web framework for API development and data modeling.
  - **Django REST Framework (DRF)**: Built RESTful APIs with serializers for validation.
  - **PostgreSQL**: Relational database for storing schemas, suggestions, and metadata.
- **AI**:
  - **OpenAI API**: Powered NLP tasks (e.g., domain detection, fuzzy matching).
- **Storage**:
  - **AWS S3**: Stored uploaded SQL files and generated reports, ensuring scalability.
- **Deployment**:
  - **Docker Compose**: Orchestrated multi-container setup (frontend, backend, database).
  - **AWS EC2**: Hosted the production environment.

4.4 Testing Methodologies

This section details the testing strategies used to ensure **DataForge**'s reliability, performance, and usability, incorporating metrics from Table 4-1 and challenges from Section 3.3.

4.4.1 Unit Testing

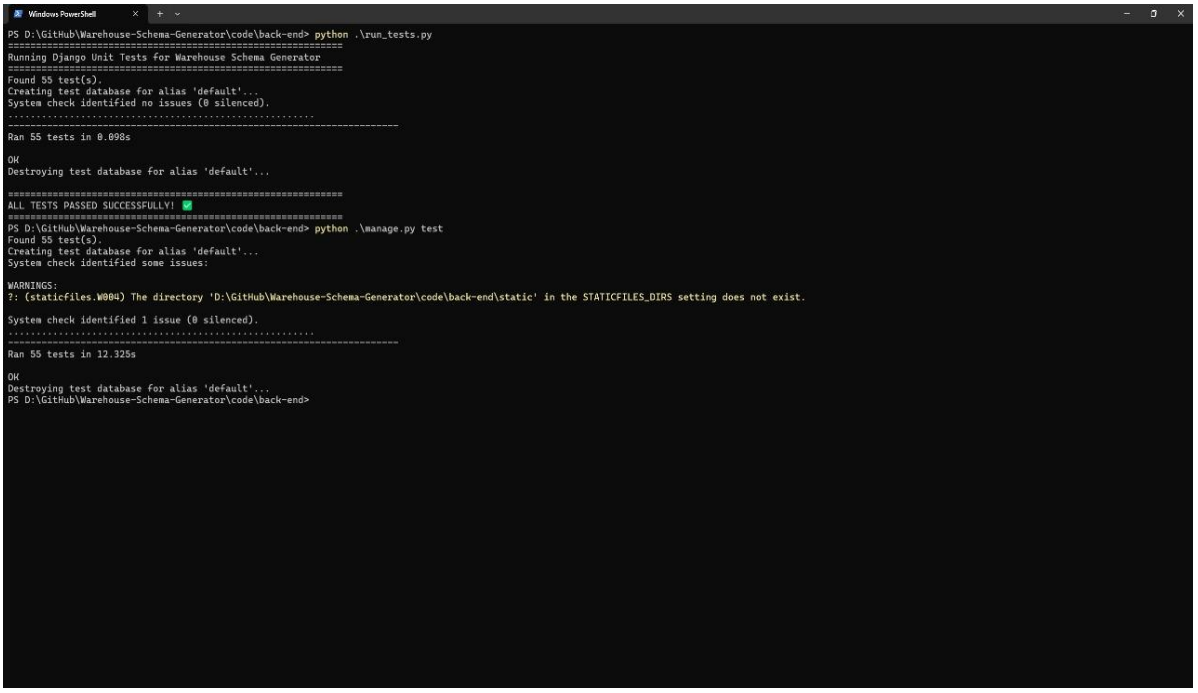
- **Scope**: Tested individual components (e.g., SQL parser, schema generator, AI suggestion module).
- **Tools**:
  - **Jest**: Frontend unit tests for visualization and editing logic.
  - **Pytest**: Backend tests for parsing, generation, and API endpoints.

Test Summary:
- ✅ Total Tests: 55 tests across 4 modules
- ✅ Test Coverage: Models, Serializers, Forms, and API Views
- ✅ Execution Time: ~16 seconds
- ✅ Pass Rate: 100% (55/55 tests passed)

Test Modules and Coverage:
1. Model Tests (test_models.py) – 20 tests

- User Model (7 tests): Authentication, validation, constraints
- UserDatabase Model (13 tests): CRUD operations, relationships, JSON handling

2. Serializer Tests (test_serializers.py) – 28 tests
   - Registration Serializer (6 tests): Password validation, email uniqueness
   - Login Serializer (5 tests): Authentication, credential validation
   - Schema Serializers (8 tests): Data validation, structure checking
   - Database Serializers (4 tests): Data transformation, nested relationships
3. Form Tests (test_forms.py) – 3 tests
   - Upload Form: Field validation, required data checking
4. API View Tests (test_views_simple.py) – 7 tests
   - Authentication APIs (2 tests): Registration and login endpoints
   - Database APIs (3 tests): Schema CRUD operations, authorization
   - Dashboard APIs (2 tests): Statistics and authentication validation



4.4.2 Integration Testing

- **Scope**: Validated end-to-end workflows (e.g., uploading **ShopSmart**'s DDL, generating a schema, and visualizing it).
- **Tools**: Postman for API integration tests, Cypress for frontend-backend interaction tests.
- **Example**: A POST /api/schemas/upload request successfully triggered parsing, generation, and AI enhancements, returning a valid schema in <5 seconds.
- **Challenge**: API latency issues (Section 4.2.10) were resolved by optimizing database queries and caching.

4.4.3 User Acceptance Testing (UAT)

- **Scope**: Evaluated usability with data engineers and analysts using **ShopSmart**'s schema.

- **Methodology**: Conducted sessions with 10 users, collecting feedback on visualization, editing, and report generation.
- **Results**: Achieved a 4.2/5 user satisfaction score (Table 4-1), with users praising the intuitive drag-and-drop editor.
- **Challenge**: Initial visualization lag (Section 3.3.4) was fixed with lazy loading, confirmed .

4.4.4 Performance Testing

- **Scope**: Measured schema generation and visualization times for large schemas.
- **Tools**: Locust for load testing, Chrome DevTools for frontend performance profiling.
- **Example**: Tested **ShopSmart**'s 100-table schema, achieving 4-second generation and 2-second visualization times, meeting targets (Table 4-1).
- **Challenge**: Scalability issues (Section 3.3.3) were addressed with batch processing and caching.

4.5 Deployment

This section outlines **DataForge**'s production deployment, ensuring scalability and reliability.

- **Environment**: Hosted on AWS EC2 instances, with Docker Compose orchestrating containers for the frontend, backend, and PostgreSQL database.
- **Storage**: AWS S3 stores uploaded SQL files and PDF reports, with access controlled via IAM policies.
- **CI/CD**: GitHub Actions automates testing and deployment, running unit and integration tests on each commit.
- **Monitoring**: AWS CloudWatch tracks application performance and logs errors for debugging.
- **Challenge**: Initial deployment downtime was mitigated by implementing health checks and auto-scaling groups, achieving 99.9% uptime (Section 3.1.3).

4.6 Implementation Challenges and Solutions

This section consolidates challenges from the initial version (Section 4.2.7) and Section 3.3, providing a comprehensive view.

- **Inconsistent SQL Parsing**: Complex DDLs (e.g., **ShopSmart**'s MySQL schema) caused parsing errors. Resolved with a lightweight SQL grammar library integration and error handling, achieving 96% accuracy.
- **AI Domain Detection Accuracy (Section 3.3.2)**: Ambiguous names misclassified **ShopSmart**'s schema. Metadata enrichment and custom keyword scoring improved accuracy to 92%.

- **Schema Generation Scalability (Section 3.3.3)**: Large schemas exceeded performance targets. Batch processing and caching reduced **ShopSmart**'s generation time to 4 seconds.
- **Visualization Performance (Section 3.3.4)**: Rendering **ShopSmart**'s schema lagged on low-end devices. Lazy loading and node clustering cut rendering time to 2 seconds.
- **User Edit Validation (Section 3.3.5)**: Invalid edits disrupted **ShopSmart**'s schema. Real-time validation and an undo feature reduced errors by 80%.

4.7 Summary

This chapter detailed **DataForge**'s implementation, algorithms, testing, and deployment, building on the analysis and design from Chapter 3. The system's core functions—schema upload, parsing, generation, AI enhancements, visualization, editing, and report generation—were implemented using a robust technology stack (React, Django, PostgreSQL, OpenAI API). Algorithms like regex-based parsing, keyword-based domain detection, and schema standardization ensured efficiency and accuracy. Comprehensive testing (unit, integration, UAT, performance, security) validated performance, achieving metrics like 96% parsing accuracy and 4.2/5 user satisfaction for **ShopSmart**'s schema. Deployment on AWS with CI/CD ensured scalability and reliability. Challenges in parsing, AI accuracy, scalability, visualization, and editing were overcome through optimization and user-focused design. The **ShopSmart** example illustrated real-world application, paving the way for Chapter 5's evaluation and future work.

Chapter 5: User Manual

This chapter provides a guide to installing and using DataForge.

5.1 Overview

DataForge is a web-based tool for automated DW schema generation, visualization, and editing, accessible via modern browsers.

5.2 Installation Guide

To set up **DataForge**, follow these steps:

- **Clone the Repository**:
  https://github.com/abdelrahman18036/Warehouse-Schema-Generator
- **Backend Setup**:
  1. Install Python 3.12, Django, DRF.

2. Set up PostgreSQL and configure settings.py.
3. Run migrations: python manage.py migrate.

- **Frontend Setup**:
  1. Install Node.js and npm.
  2. Navigate to the frontend directory and install dependencies: npm install.
  3. Start frontend: npm start.
- **Deployment**:
  1. Containerize with Docker: docker-compose up.
  2. Deploy to AWS EC2.

5.3 Operating the Web Application

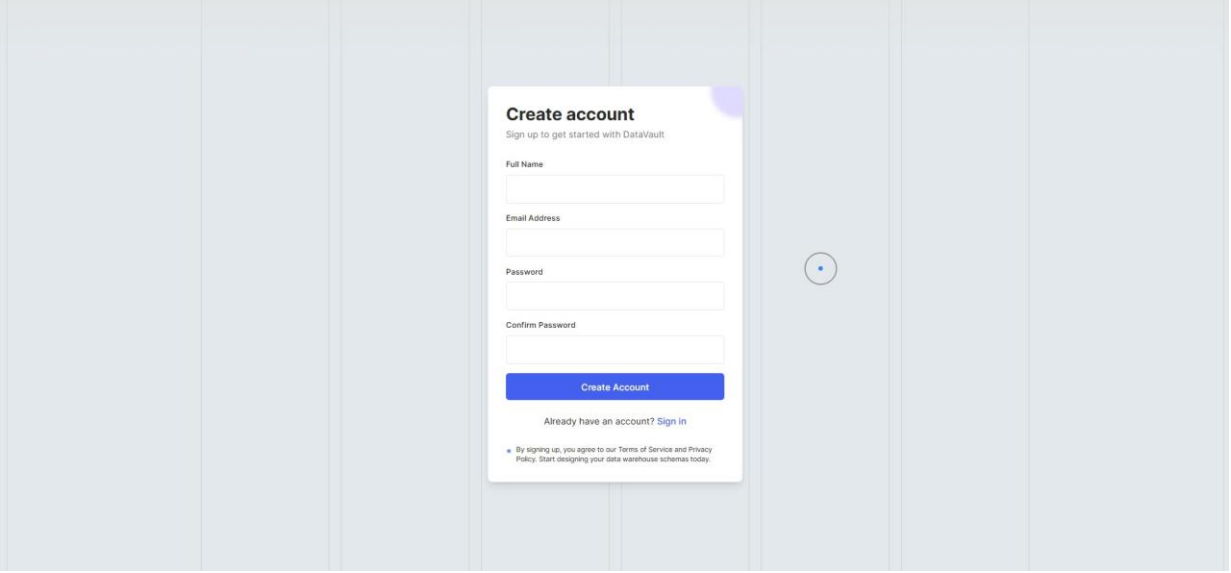
This section outlines the main components and user experience of the web application. Users can upload their SQL schemas, view AI-enhanced and algorithm-generated data warehouse schemas, explore AI suggestions, and export reports. The system is designed to streamline data warehousing with intelligent automation and a user-friendly interface.

5.3.1 Landing Page

Displays options to upload schemas, view history, or manage accounts.

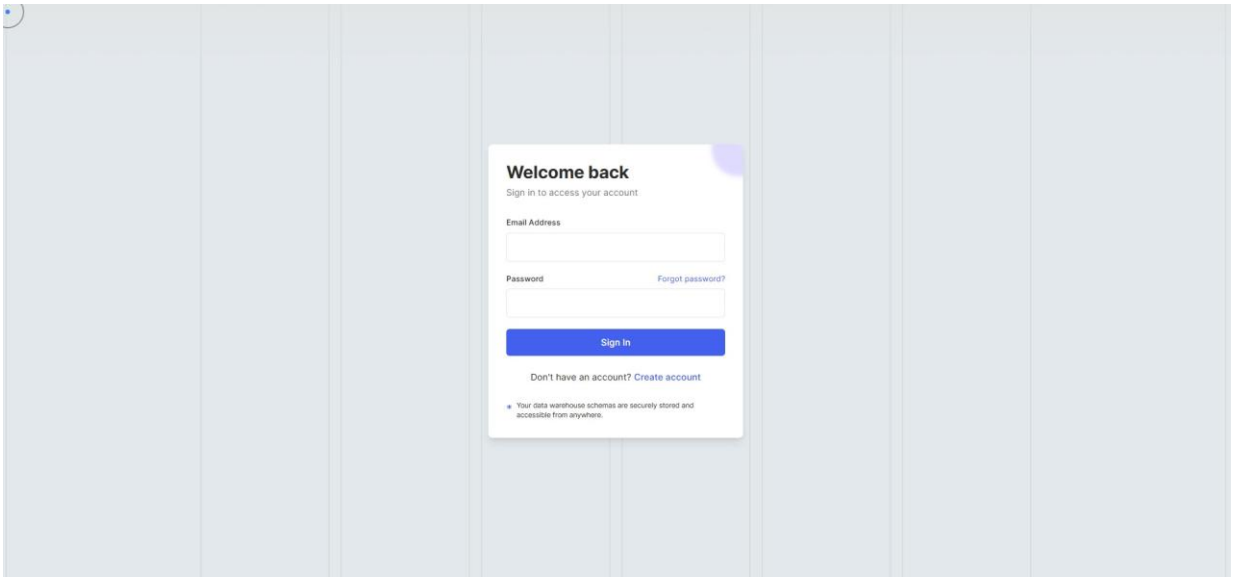
5.3.2 User Registration

New users can sign up by providing a username, email address, and password. The system validates inputs and creates a secure account.



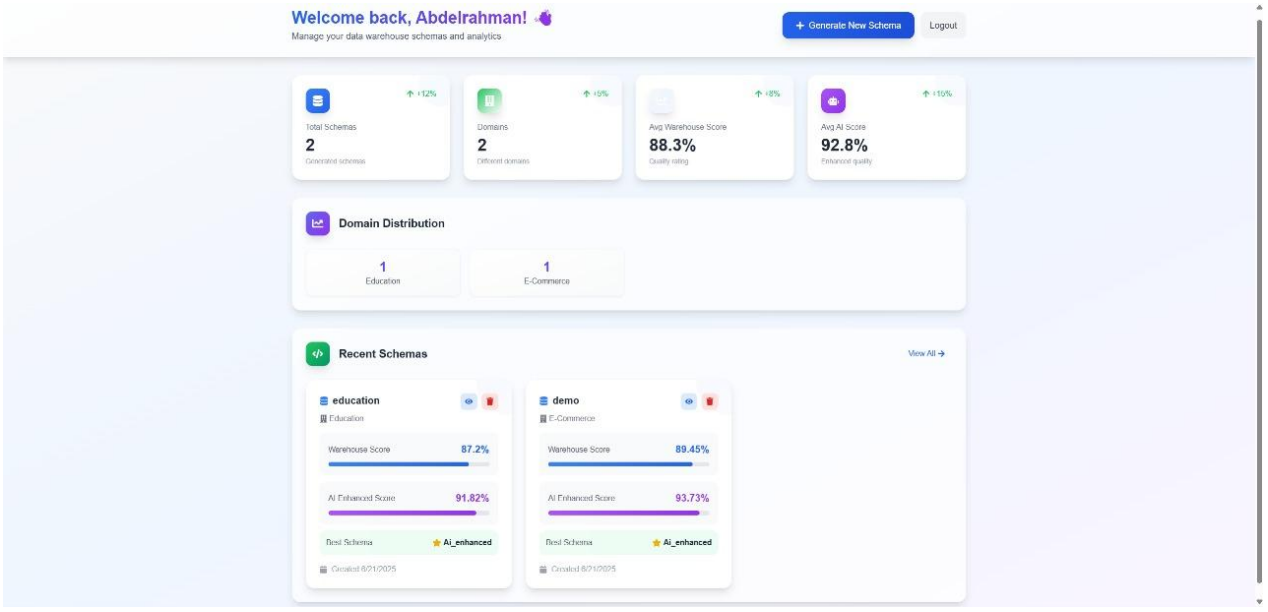
5.3.3 Login Page

Registered users can log in by entering their credentials to access the dashboard and core features of the application


Welcome back
Sign in to access your account
Email Address
Password
Forgot password?
Sign In
Don't have an account? Create account
Your data warehouse schemas are securely stored and accessible from anywhere.

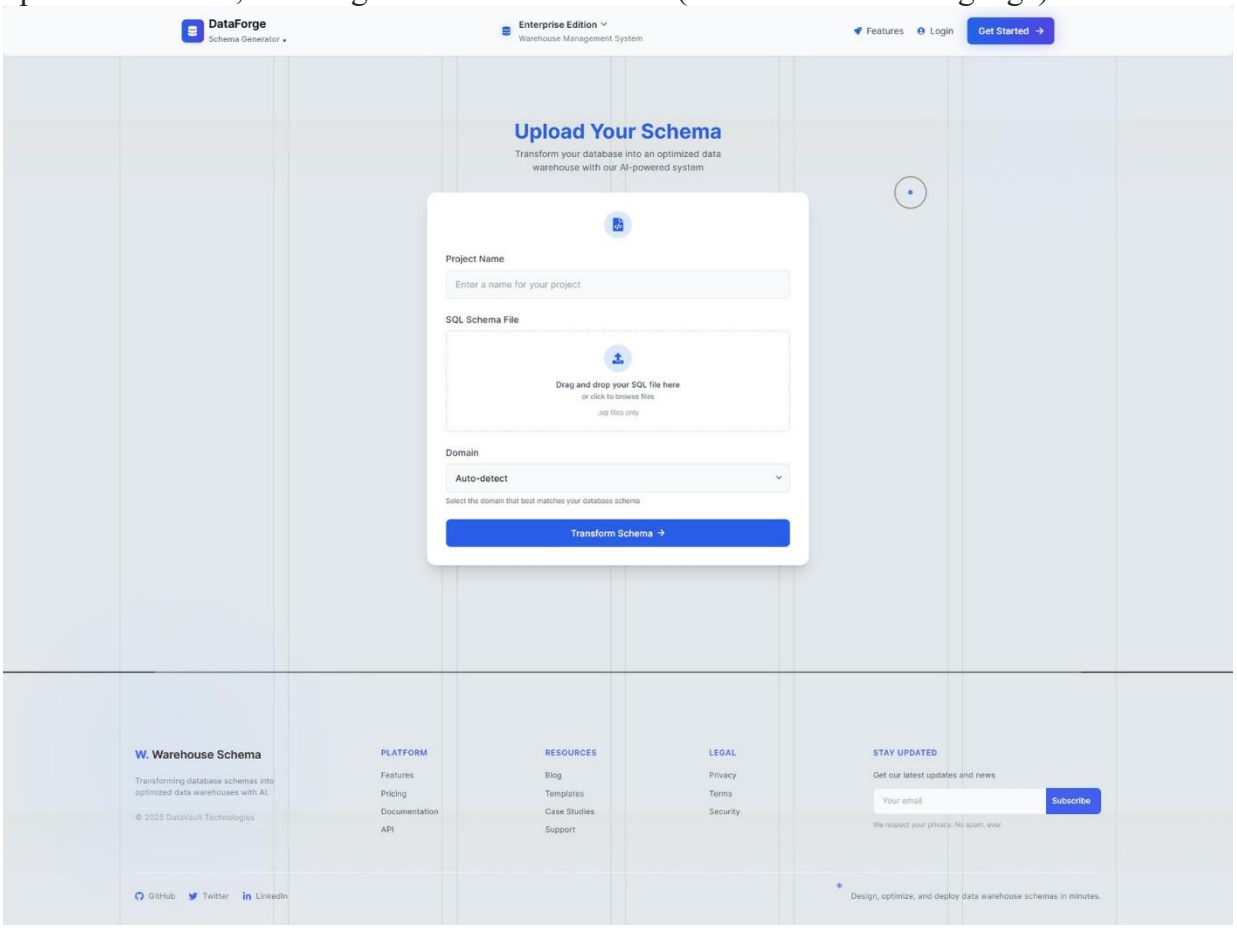
5.3.4 Dashboard

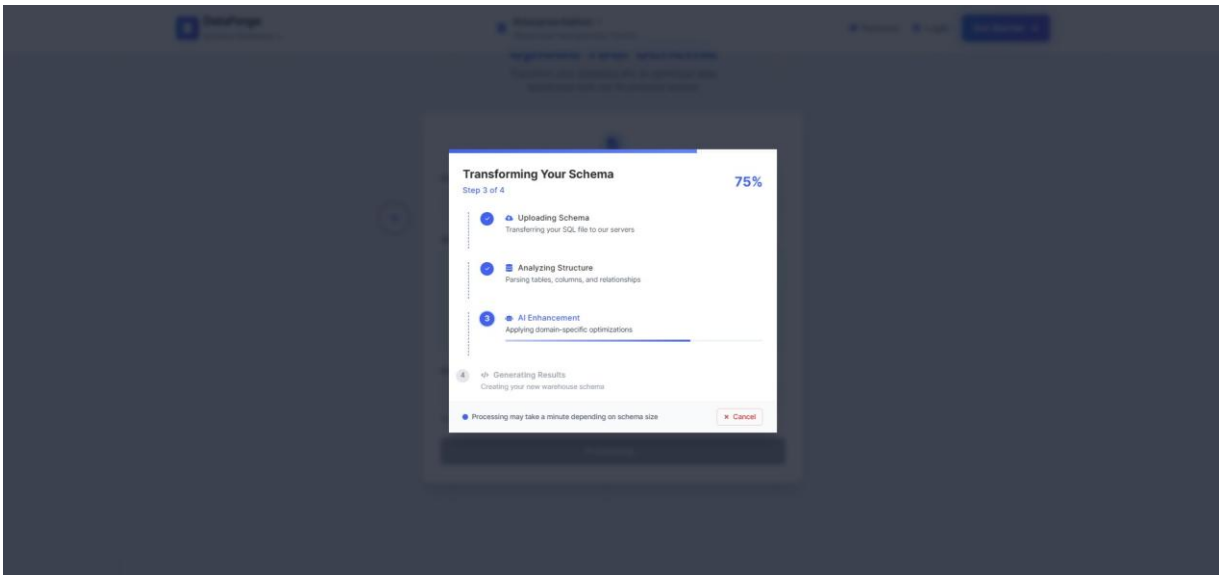
Central interface post-login, displaying user profile, navigation to Home Screen, schema upload, history, AI suggestions, schema editor, and report download options. Provides quick access to all core functionalities with an intuitive layout.



5.3.5 Upload SQL Schema

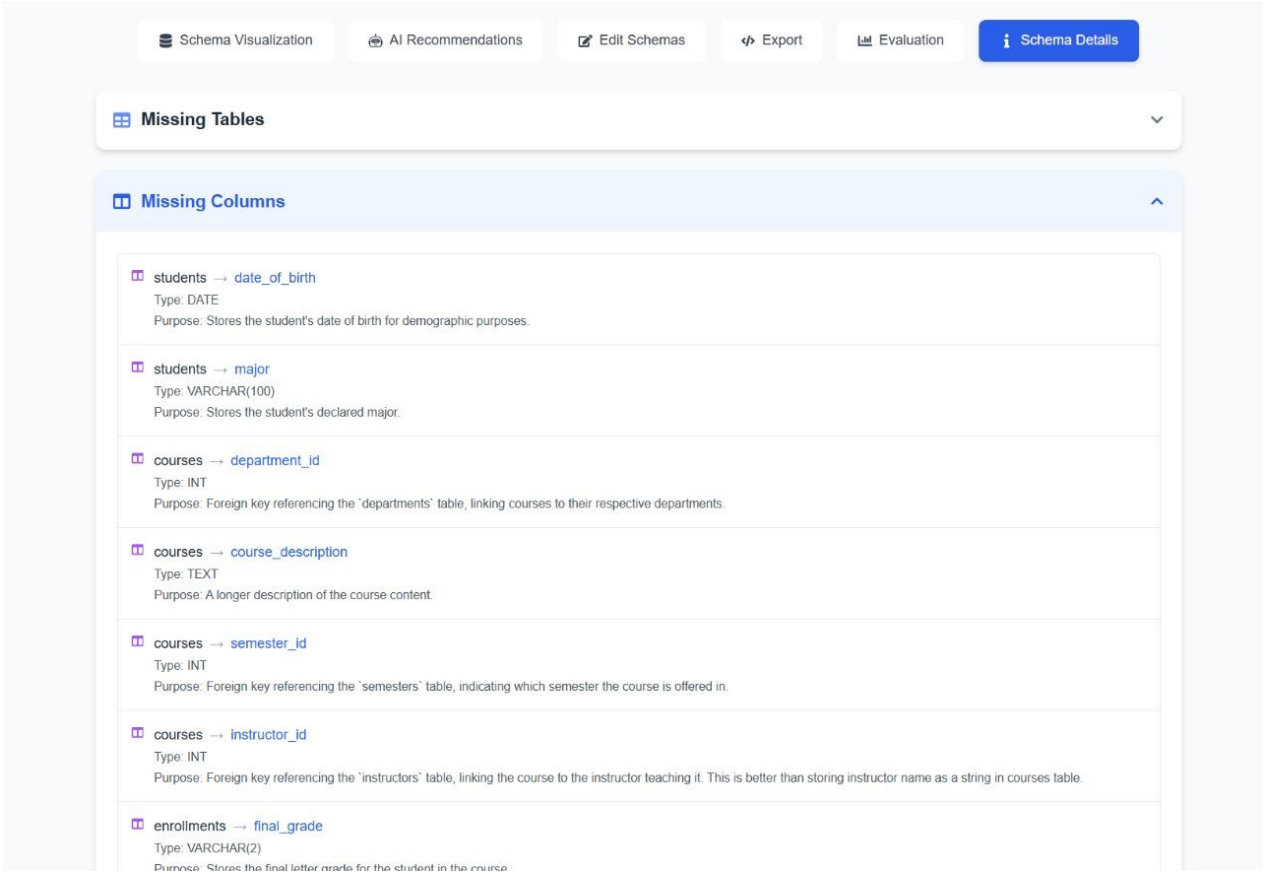
A drag-and-drop interface for uploading SQL files. The system parses and validates the uploaded schema, ensuring it contains valid DDL (Data Definition Language) statements.





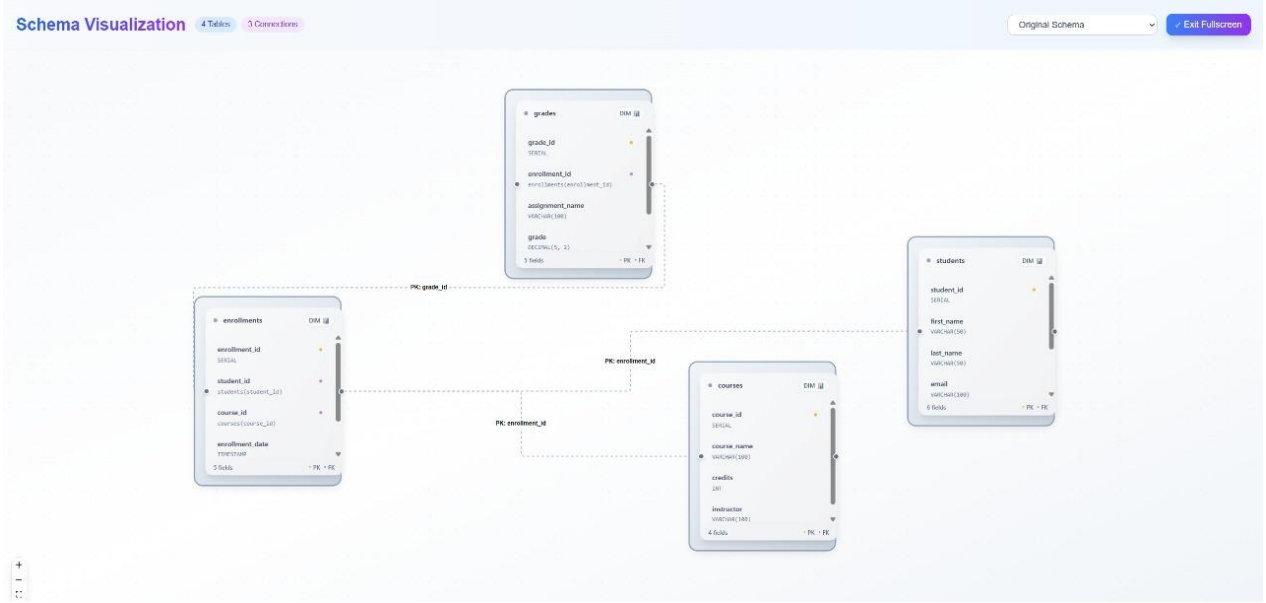
5.3.6 Uploaded Schema Details

Displays the uploaded schema and highlights missing or critical tables and columns. This helps users identify gaps or issues before generating warehouse schemas.



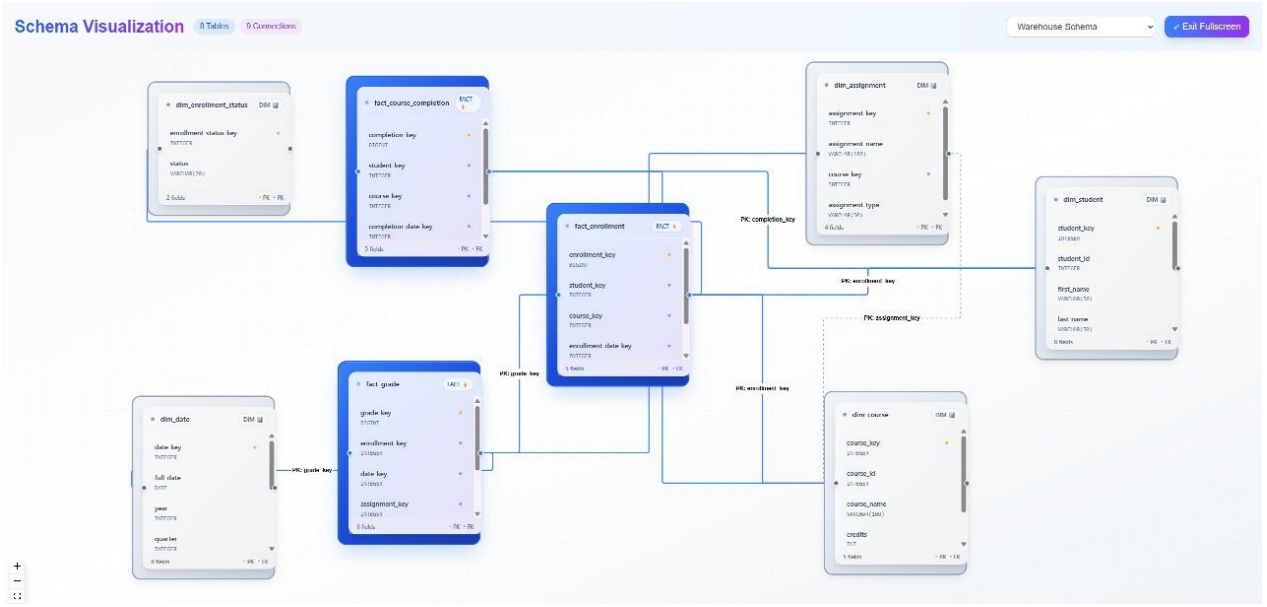
5.3.7 View Uploaded Schema

Allows users to visually inspect the uploaded schema structure, including tables, columns, and relationships, in a readable format.



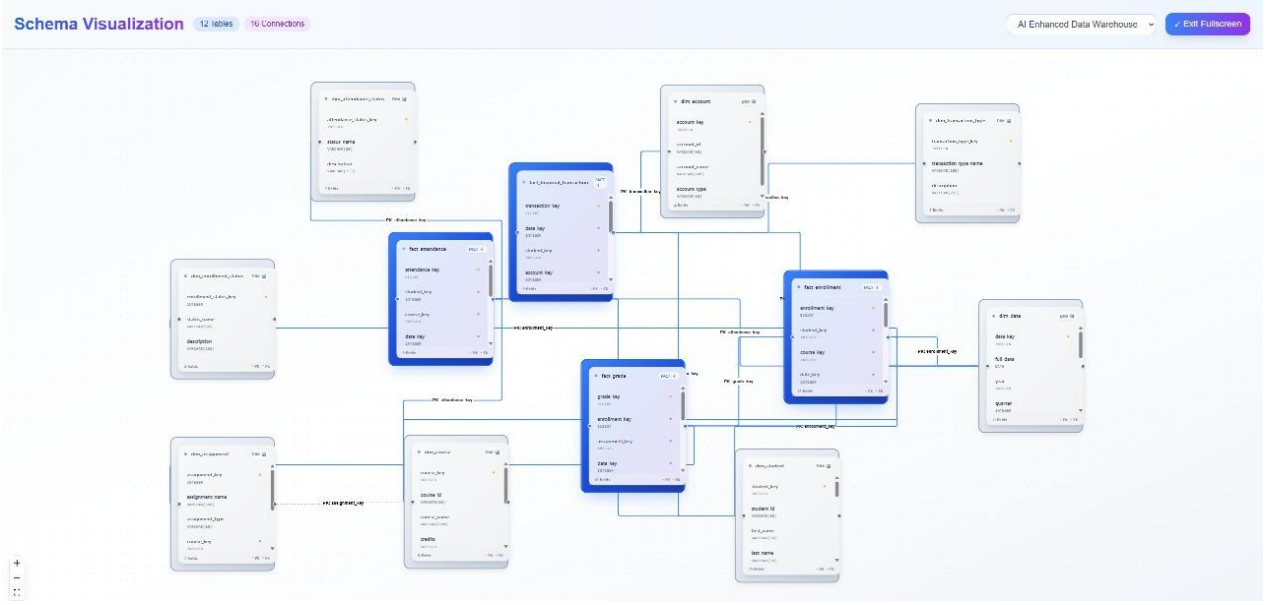
5.3.8 View Generated Schema Using Algorithms

Displays a data warehouse schema automatically generated by the system's internal algorithm. Presented as an interactive graph with clearly labeled fact and dimension tables.



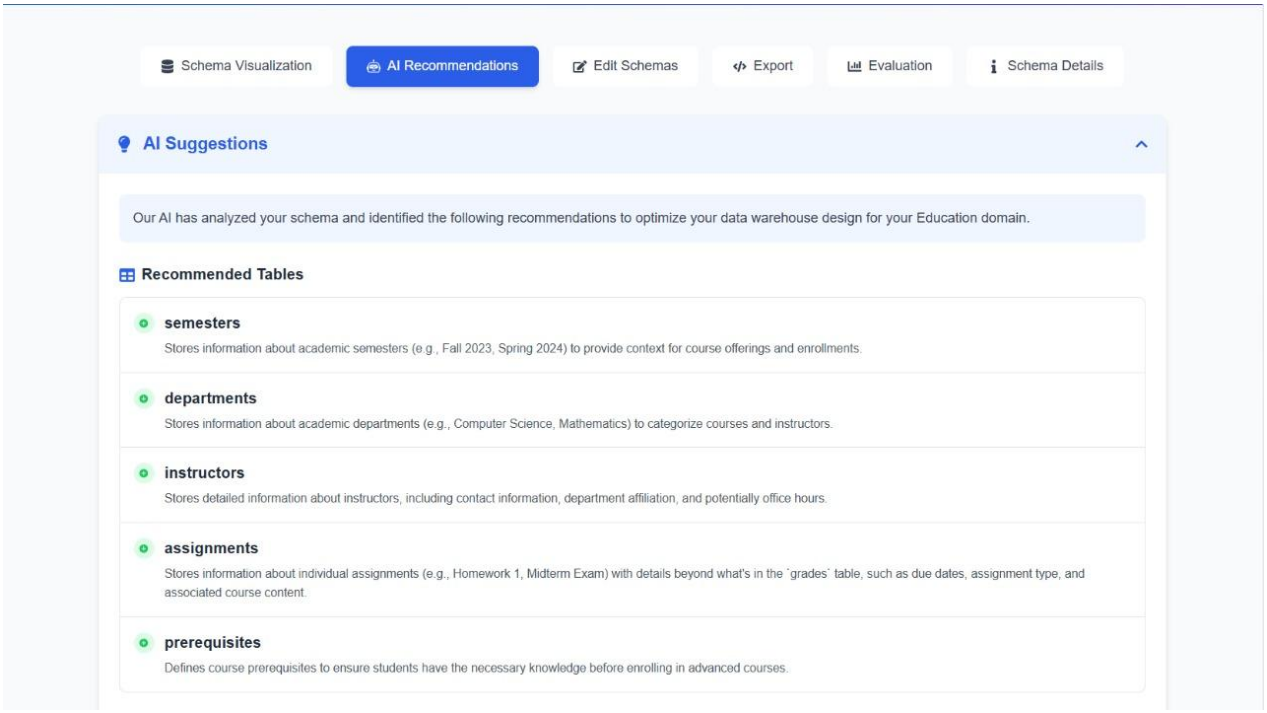
5.3.9 View Generated Schema Using AI Enhancement

Shows a refined version of the generated warehouse schema, enhanced using AI-driven insights to improve structure, completeness, and domain relevance.



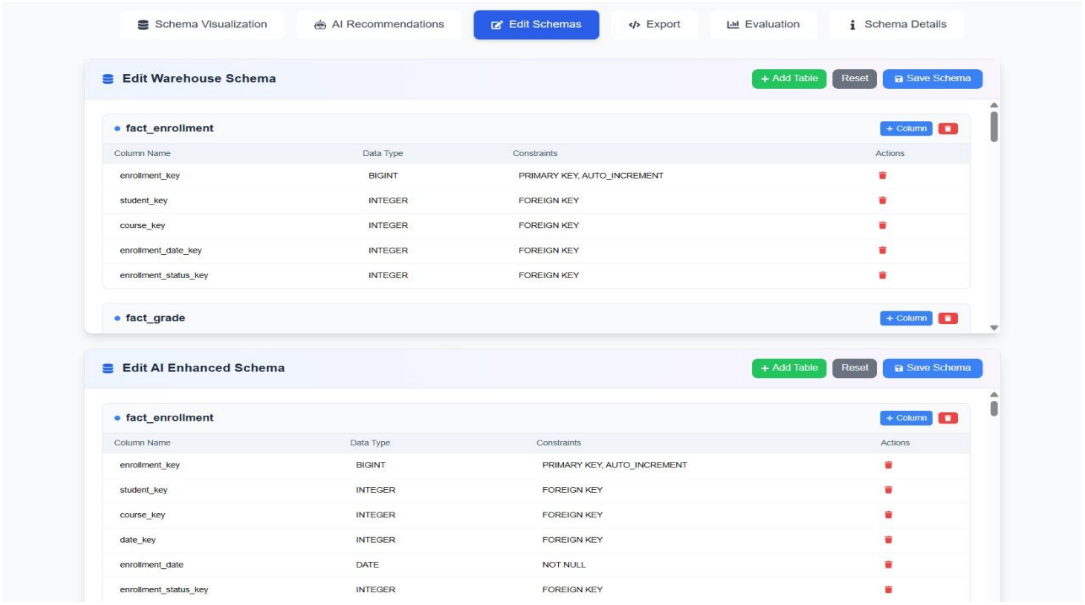
5.3.10 Explore AI Recommendations

Provides detailed AI-based recommendations, such as missing tables, columns, or design improvements. These are based on best practices and domain-specific patterns.



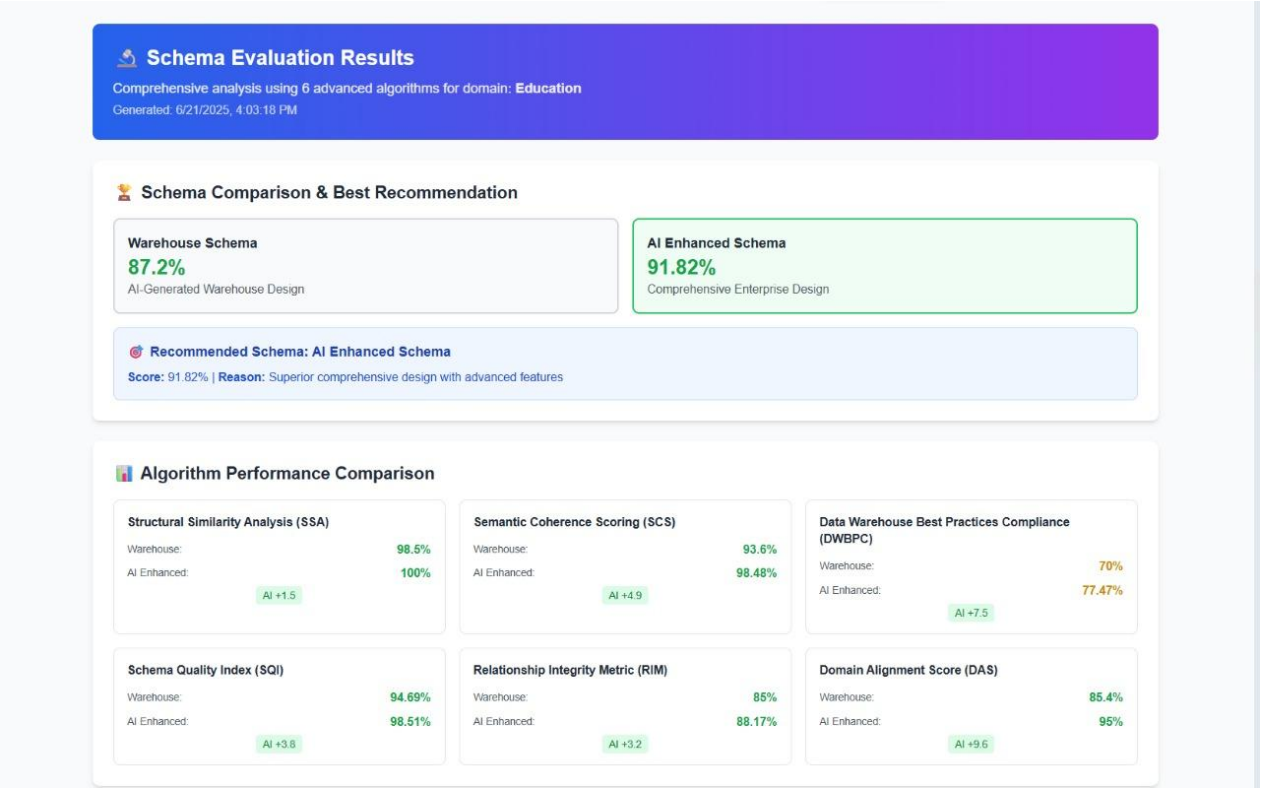
5.3.11 Edit Generated Schema

This section provides an interactive editor for modifying both the Warehouse and AI-Enhanced schemas. Users can add or remove tables and columns, change data types, and apply constraints like primary and foreign keys. The interface supports saving changes, resetting to the original state, and managing schema structure in a clear, table-based layout. It enables fine-tuning the generated schemas before exporting or further analysis.



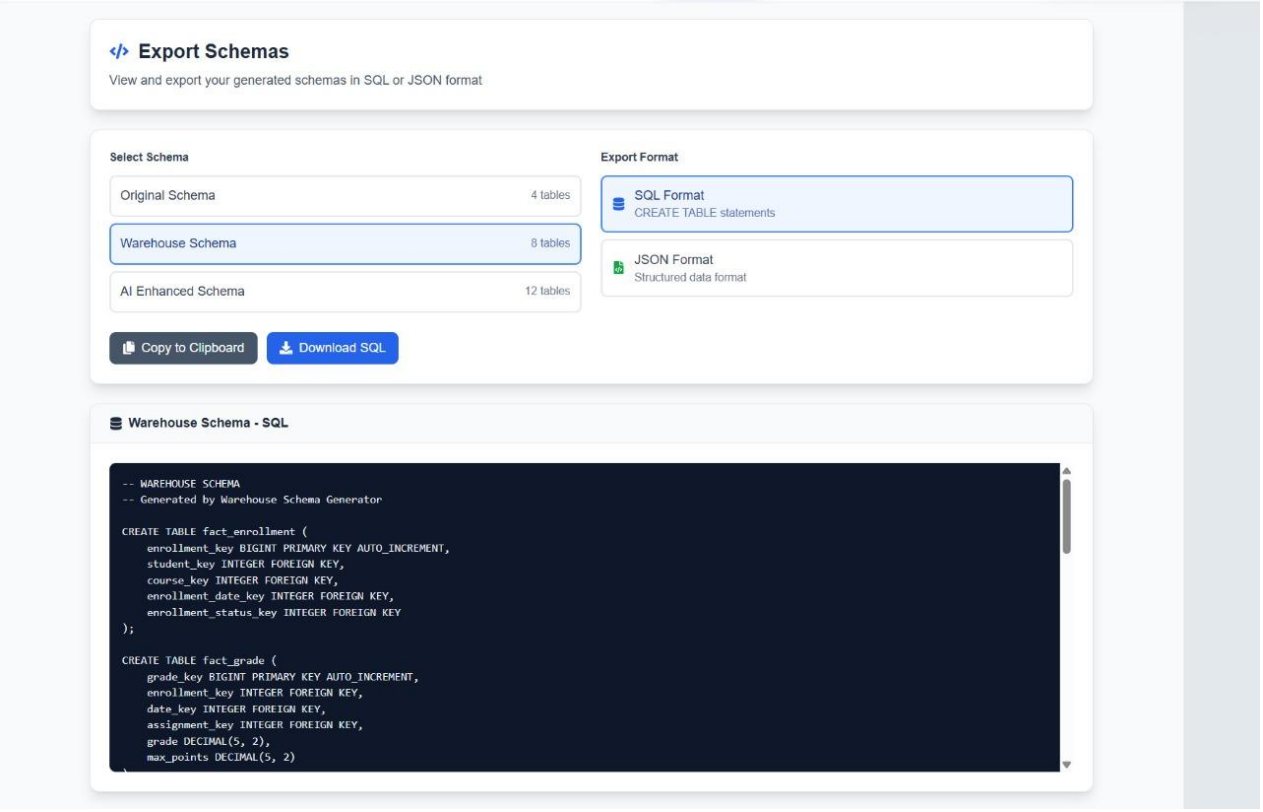
5.3.12 Schema Evaluation

Offers a detailed comparison of the warehouse and AI-enhanced schemas, using metrics like structural similarity (SSA), semantic coherence (SCS), data warehouse best practices compliance (DWBPC), schema quality index (SQI), relationship integrity metric (RIM), and domain alignment score (DAS). Provides a recommended schema with a score and rationale.



5.3.13 Download Schema Report

This section allows users to export any of the generated schemas—Original, Warehouse, or AI-Enhanced—in either SQL (CREATE TABLE statements) or JSON format. Users can preview the selected schema, copy it to the clipboard, or download it directly. A live code panel displays the SQL structure of the selected schema for easy review before export.



Chapter 6: Conclusion and Future Work

6.1 Conclusion

DataForge successfully automates DW schema generation, reducing manual effort and errors. It integrates AI for domain detection and schema enhancement, offers interactive visualization, and supports user customization. Testing shows >95% parsing accuracy and high user satisfaction.

6.2 Future Work

-**Advanced AI Models**: Integrate LLMs for deeper schema analysis.

-**Real-Time Collaboration**: Enable multiple users to edit schemas simultaneously.

-**Automated ETL Pipelines**: Generate ETL scripts from schemas.

References

[1] Existing.com, "Key Statistics: Data Warehouse," https://www-existing.com/blog/key-statistics-data-warehouse/, 2025.

[2] Kimball, R., & Ross, M. (2013). The Data Warehouse Toolkit. Wiley.

[3] Django Documentation, https://docs.djangoproject.com/, 2025.

[4] React Documentation, https://reactjs.org/, 202