



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 بالتعبير 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، يقدم DataForge مخططات عالية الجودة وثابتة ومتوافقة مع مجال البيانات في جزءٍ يسير من الوقت المعتاد. بالإضافة إلى ذلك، تمكن 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، يمهد هذا المشروع الطريق لنموذج جديد في هندسة البيانات المؤتمتة—حيث يصبح تصميم المخطط ليس سريعاً ودقيقاً فحسب، بل ذكياً، قابلاً للتكيف، ومتكاملاً بسلسلة في دورة حياة التحليل.

Table of Contents

Acknowledgement	ii
Abstract	iii
Arabic Abstract	iv
Table of Contents	v
List of Figures	viii
LIST OF ABBREVIATIONS	X
CHAPTER ONE: INTRODUCTION	12
1.1 Preface	12
1.2 Significance and Motivation	13
1.3 Problem Definition	13
1.4 Aims and Objectives	14
1.5 Methodology	15
1.6 Timeline	16
1.7 Time Plan	17
1.8 Thesis Outline	18
CHAPTER TWO: LITERATURE REVIEW	19
2.1 Introduction	19
2.2 Theoretical Background	20
2.3 Previous Studies and Works	30
CHAPTER THREE: SYSTEM ARCHITECTURE AND METHODS	34
3.1 System Architecture	34
3.2 Methods and Procedures	39
3.3 Functional Requirements	44
3.4 Nonfunctional Requirements	45
3.5 System Analysis & Design	46
3.6 Development Challenges and Solutions	50
CHAPTER FOUR: SYSTEM IMPLEMENTATION AND RESULTS	52
4.1 Materials and Environment	52

4.1.1 Datasets	52
4.1.2 Software and Frameworks	53
4.1.3 Hardware & Cloud Configuration	53
4.2 Implementation Details	54
4.2.1 Schema Upload & Preprocessing	54
4.2.2 SQL Parsing	54
4.2.3 Heuristic Schema Generation	55
4.2.4 AI-Driven Enhancements	55
4.2.5 Visualization & Editing	55
4.2.6 Export & Reporting	55
4.3 Experimental Results	56
4.3.1 Hypotheses	56
4.3.2 End-to-End Metrics	56
4.3.3 Statistical Summaries	56
4.3.4 Negative Findings	56
4.3.5 Algorithmic Evaluation	57
4.4 Testing Methodologies	60
4.4.1 Unit Testing	60
4.4.2 Integration Testing	61
4.4.3 Performance Testing	61
4.5 Deployment	61
4.6 Implementation Challenges & Solutions	61
CHAPTER FIVE: RUN THE APPLICATION	62
5.1 Overview	62
5.2 Installation Guide	62
5.3 Operating the Web Application	63
5.3.1 Landing Page	63
5.3.2 User Registration	64
5.3.3 Login Page	64
5.3.4 Dashboard	65
5.3.5 Upload SQL Schema	65
5.3.6 Uploaded Schema Details	66
5.3.7 View Uploaded Schema	67
5.3.8 View Generated Schema Using Algorithms	67
5.3.9 View Generated Schema Using AI Enhancement	68
5.3.10 Explore AI Recommendations	68
5.3.11 Edit Generated Schema	69
5.3.12 Schema Evaluation	69
5.3.13 Download Schema Report	70
CHAPTER SIX: CONCLUSION AND FUTURE WORK	71
6.1 Conclusions	71
6.2 Significance and Practical Implications	72
6.3 Limitations and Misconceptions	72

6.4 Future Work and Recommendations _____ **73**

REFERENCES _____ **74**

List of Figures

● Figure 1-1: Project Time Plan.....	16
● Figure 2-1: Data Warehouse and Business Intelligence System Architecture.....	20
● Figure 2.2: Fact Table Structure	22
● Figure 2.3: Overwrite outdated Type 1.....	23
● Figure 2.4: Slowly Changing Dimension Type 2.....	23
● Figure 2.5: Add a new column for old values Type 3.....	24
● Figure 2.6: Retail Sales Star Schema.....	24
● Figure 2.7: Data Warehouse Bus Matrix.....	26
● Figure 2.8: ETL Process Flow.....	27
● Figure 2.9: DDL for ShopSmart.....	28
● Figure 3.1: DataForge System Architecture Diagram.....	34
● Figure 3.2: Use Case Diagram.....	46
● Figure 3.3: Class Diagram.....	47
● Figure 3.4: Sequence Diagram.	48
● Figure 3.5: Database Diagram.....	49
● Figure 4.1: SQL Parsing.....	54
● Figure 4.2: Parsing Result.....	54
● Figure 4-3: Unit Testing.....	60
● Figure 5.1: Landing Page.....	63
● Figure 5.2: Landing Page II.....	63
● Figure 5.3: User Registration.....	64
● Figure 5.4: Login Page.....	64
● Figure 5.5: Dashboard.....	65
● Figure 5.6: Upload SQL Schema.....	66
● Figure 5.7: User Feedback.....	66
● Figure 5.8: Upload Schema Details.....	67
● Figure 5.9: View Uploaded Schema.....	67
● Figure 5.10: View Generated Schema Using Algorithms.....	68
● Figure 5.11: View Generated Schema Using AI Enhancement.....	69
● Figure 5.12: Explore AI Recommendations.....	69
● Figure 5.13: Edit Generated Schema.....	70
● Figure 5.14: Schema Evaluation.....	71
● Figure 5.15: Download Schema Report.....	72

List of Tables

Table 1-1: Project Time Plan.....	17
Table 2-1: Schema Design Patterns.....	21
Table 2-2: 3NF vs. Dimensional Modeling.....	21
Table 2-3: Sample Date Dimension.....	22
Table 2-4: Procedure And Usecases.....	24
Table 3-1: Functional Requirements.....	44
Table 3-2: Nonfunctional Requirements.....	45
Table 4-1: Datasets used, with structure and context.	52
Table 4-2: Software and libraries employed.	53
Table 4-3: Hardware and Huawei Cloud instances.	53
Table 4-4: End-to-end performance and accuracy.	56
Table 4-5: Algorithmic evaluation results.	59
Table 4-6: Challenges and resolutions.	61

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

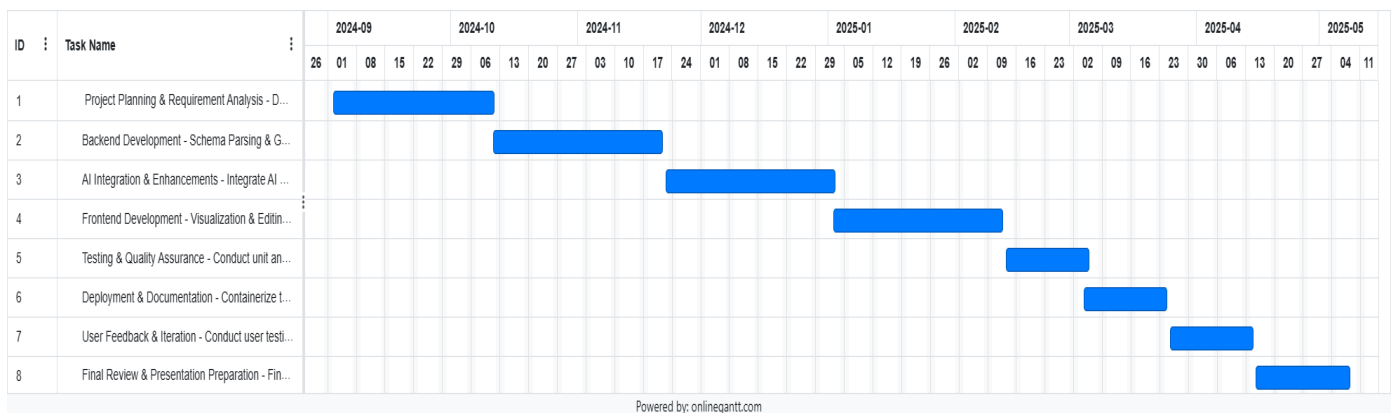


Figure 1-1: Project Time Plan

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

Table 1-1: Project Time Plan

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.

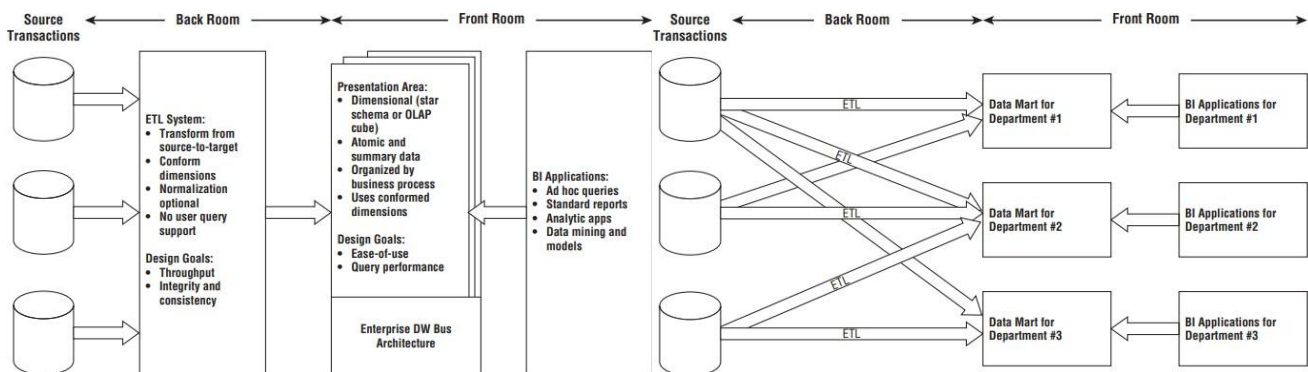


Figure 2-1: Data Warehouse and Business Intelligence System Architecture

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.

Table 2-1: Schema Design Patterns

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

Table 2-2: 3NF vs. Dimensional Modeling

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).

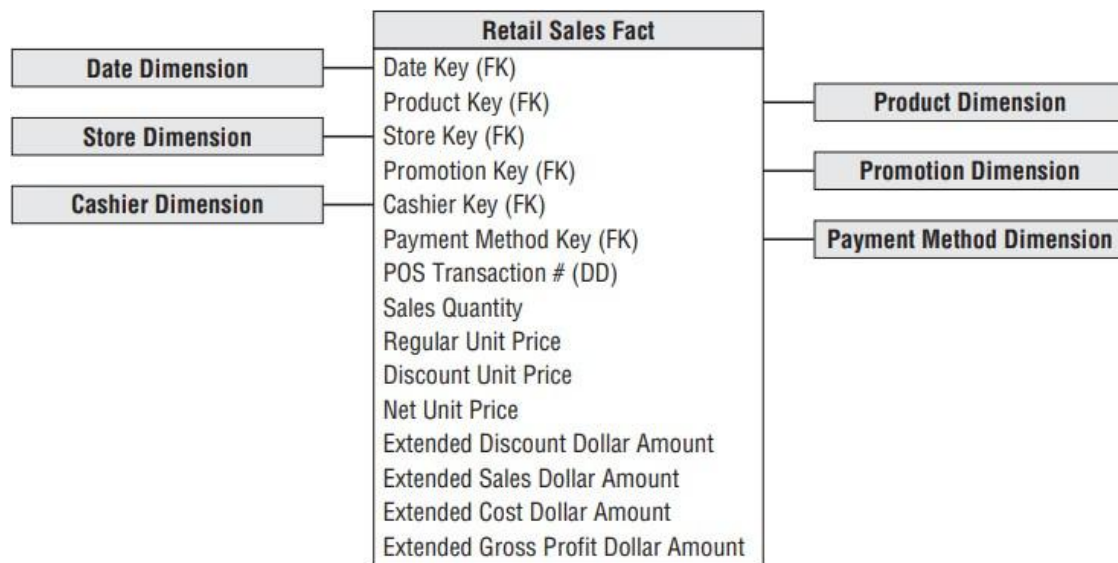


Figure 2.2: Fact Table Structure

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

Table 2-3: Sample Date Dimension

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

Table 2-4: Procedure And Usecases

Retail Sales Star Schema

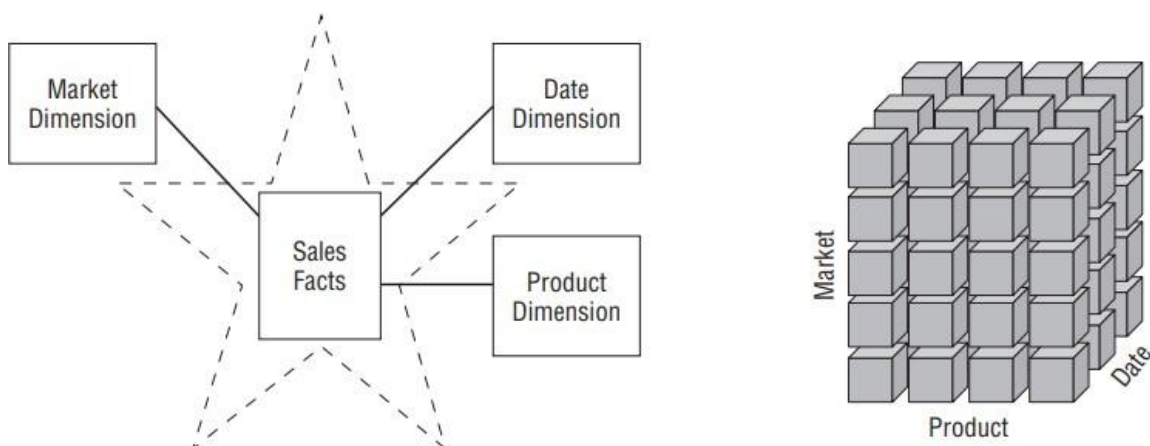


Figure 2.6: Retail Sales Star Schema

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:

BUSINESS PROCESSES	COMMON DIMENSIONS						
	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.

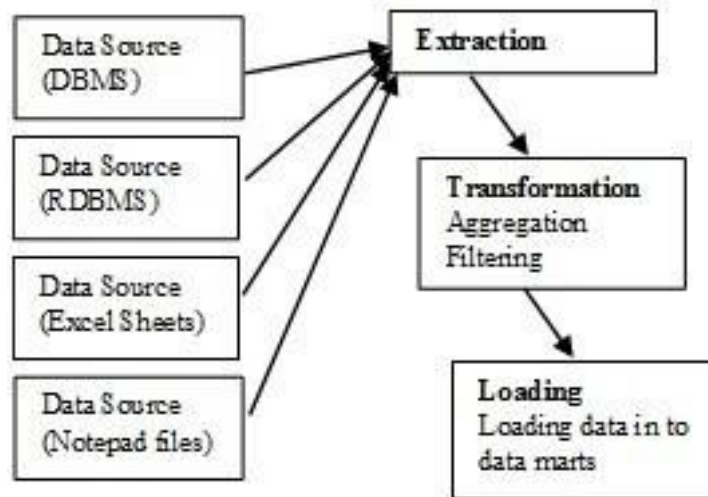


Figure 2.8: ETL Process Flow

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 2.9: 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.

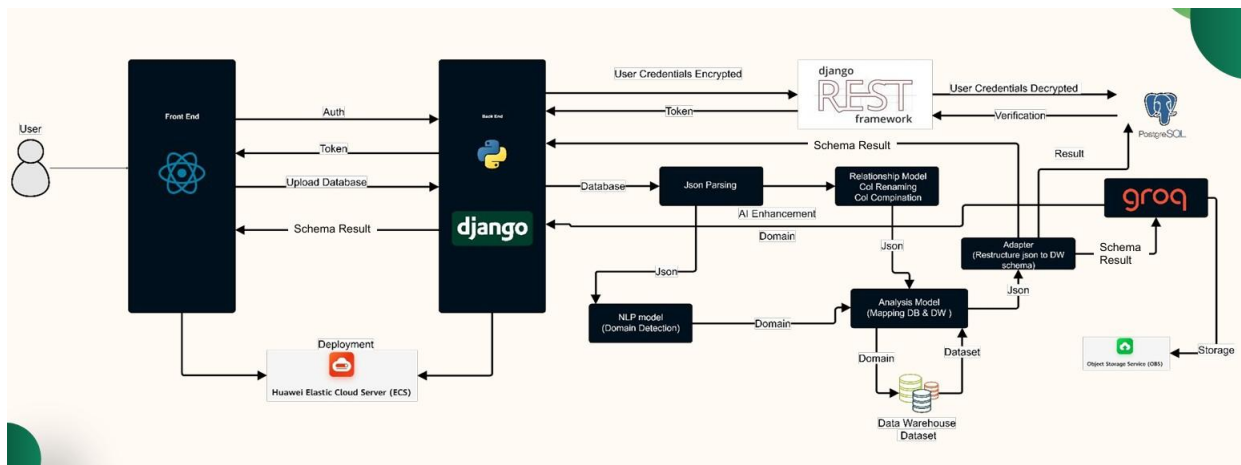


Figure 3.1: DataForge System Architecture Diagram

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.
 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.
- **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 Functional Requirements

The following table summarizes **DataForge**'s functional requirements, aligned with the project's objectives:

Requirement	Description
User Authentication	Secure account creation, login, and session management.
Schema Upload	Upload and validate SQL schema files for correct syntax, size limits, and supported dialects.
Schema Parsing	Extract table names, columns, data types, primary keys, and foreign keys from uploaded SQL.
Schema Generation	Categorize tables as fact or dimension, define primary/foreign keys, and assemble an initial schema.
AI Enhancements	Detect domains, suggest missing tables/columns, optimize schemas.
Schema Visualization	Display schemas as interactive, draggable graphs with distinct styles for fact vs. dimension tables.
Schema Editing	Allow users to add, remove, rename, or reorder tables/columns and immediately see the effects.
Versioning & Undo	Record every schema change and enable undo/redo of edits via timestamped diff history.
Export & Reporting	Download final schemas in SQL, JSON, including AI suggestions.
Metadata Management	Store and retrieve metadata (domain labels, AI suggestions, edit history) linked to each schema.
Error Handling	Provide meaningful error messages for invalid uploads or system failures.

Table 3-1: Functional Requirements

3.4 Nonfunctional Requirements

Nonfunctional requirements ensure **DataForge**'s performance and usability:

Requirement	Description
Scalability	Handle schemas with 100+ tables efficiently.
Performance	Process schemas and render visualizations in under 5 seconds for typical inputs.
Usability	Intuitive UI accessible to non-technical users.
Security	Encrypt user data and use HTTPS for secure API communication.
Reliability	Achieve 99.9% uptime and ensure accurate schema generation.
Compatibility	Support modern browsers and SQL dialects.
Maintainability	Modular code structure, high unit/test coverage, clear documentation, and coding standards enforcement.
Testability	Automated unit, integration, and end-to-end tests (using Django's test framework and <code>run_tests.py</code>).

Table 3-2: Nonfunctional Requirements

3.5 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.

3.5.1 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.

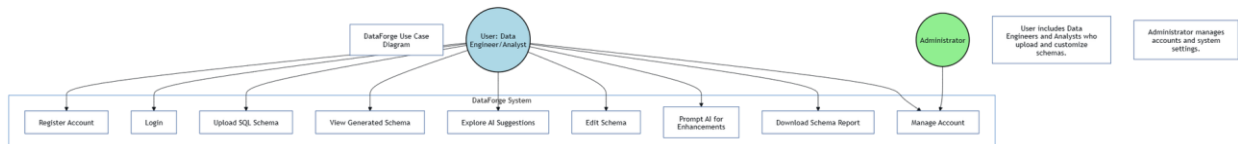


Figure 3.2: Use Case Diagram

3.5.2 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()

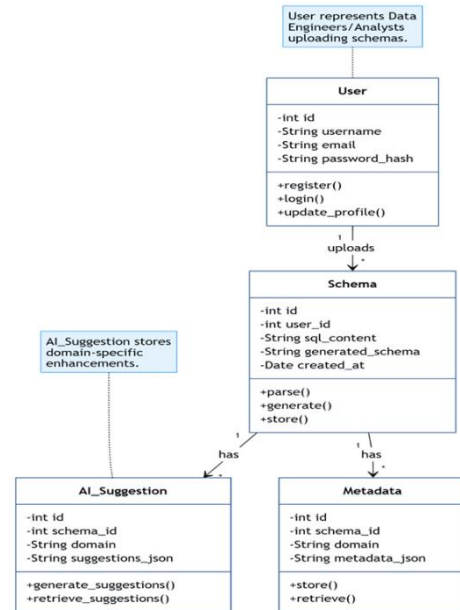


Figure 3.3: Class Diagram

- **Relationships:**

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

3.5.3 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

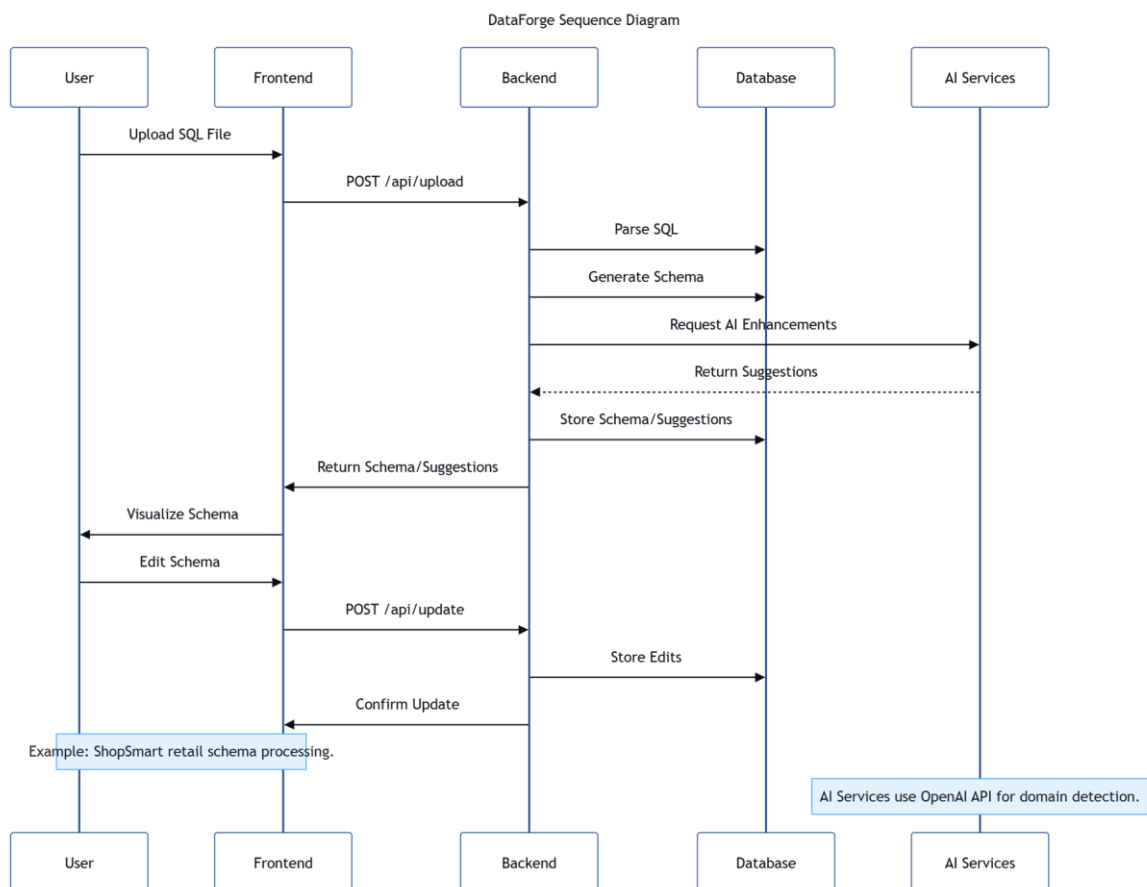


Figure 3.4: Sequence Diagram

3.5.4 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).

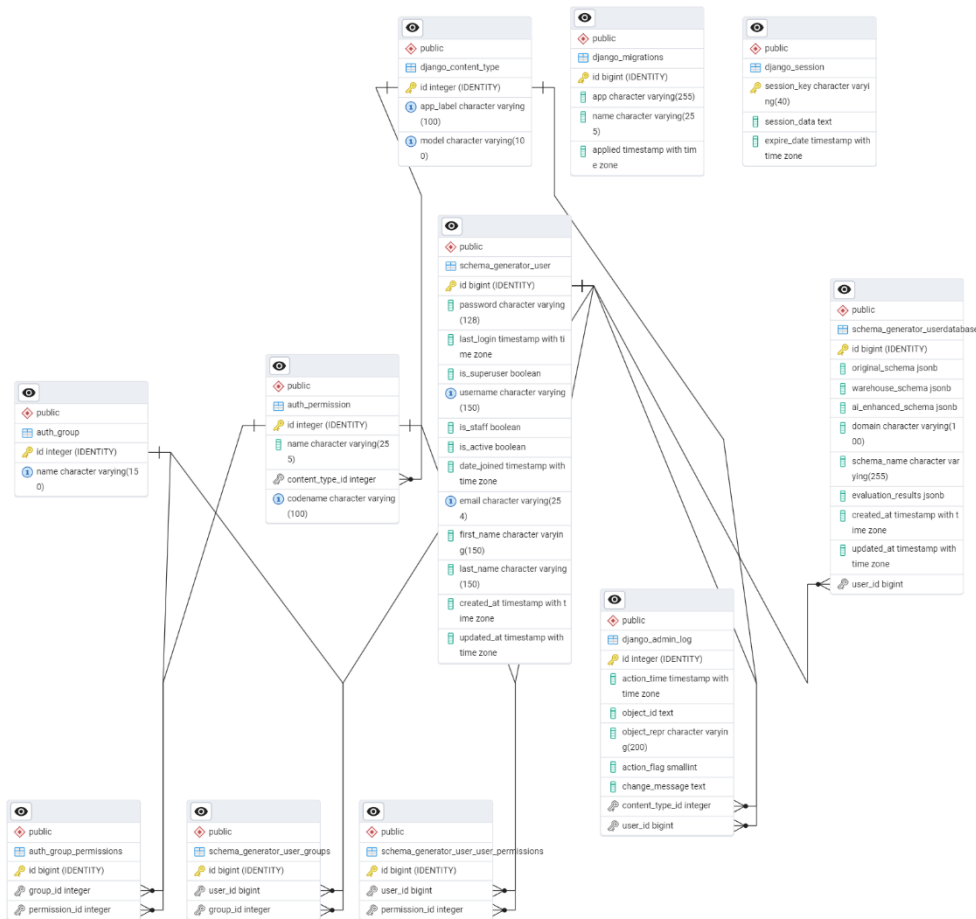


Figure 3.5: Database Diagram

3.6 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 Four: System Implementation and Results

This chapter describes the materials, environment, implementation details, experimental results, and critical analysis of DataForge. We begin with the datasets, software, and hardware used, then detail the core implementation and results.

4.1 Materials and Environment

4.1.1 Datasets

Dataset	Source & Version	Domain	Tables	Columns	Notes
AdventureWorks DW	AdventureWorksDW2008 (PDF)	Manufacturing, Sales	25	10	Microsoft sample DW; sales, purchasing, HR, inventory, and finance data.
Amazon Redshift Example	AWS Redshift Docs (p.47)	Cloud DWH/General	~14	~9	Illustrative schema for cloud data warehousing; users, events, sales.
Peru Manufacturing DWH	Kimball Sales DWH in Peru	Manufacturing	8	~7	Kimball-based star schema for sales analysis; dimensions for product, time, and region.
Forest Inventory DWH	Forest Management DWH	Environmental/Forestry	11	~6	Combines geospatial forest inventory and satellite image analysis.
Education Accreditation DWH	Higher Education DWH	Higher Education	10	~8	Supports BAN-PT accreditation and collaboration reporting (KERMA); includes student, publication, partnership data.
Pathogen Monitoring DWH	Pathogen Monitoring DWH	Healthcare/Public Health	9	~7	Tracks pathogens, regions, facilities, and mitigation actions.
Steam User Behavior DWH	Steam DW Analysis	Gaming/Behavioral Analytics	7	~6	Uses Steam API data; builds facts on playtime, achievements, purchases.
Orange Digital Center	Orange DC Internship Project	General / Internship Demo	6	~5	Educational DW project; includes customer, services, and performance logs.

Table 4-1: Datasets used, with structure and context.

4.1.2 Software and Frameworks

Component	Version	Vendor/Publisher	Purpose & Rationale
Django	4.2	Django Software Foundation	Backend MVC and REST API framework.
Django REST Framework	3.14	DRF community	JSON serialization and view routing.
React	18.2.0	Meta Platforms, Inc.	Dynamic, component-based frontend.
ReactFlow	10.0	ReactFlow community	Drag-and-drop schema graph visualization.
Tailwind CSS	3.3.2	Tailwind Labs	Utility-first styling for responsive layout.
Axios	1.4.0	Axios community	HTTP client for AJAX calls.
sqlparse	0.4.3	Python community	SQL AST for dialect normalization.
scikit-learn	1.3.0	Python community	TF-IDF, logistic regression for domain detection.
Transformers	4.33	HuggingFace	BERT embeddings for semantic similarity.
Google Gemini Flash API	—	Google	LLM-driven schema suggestion generation, Template-based optimization advice.
PostgreSQL	17.0	PostgreSQL Global Dev. Group	ACID-compliant DB with JSON support.
Docker & Docker Compose	24.0 / 2.20	Docker, Inc.	Containerization for consistency.
pytest	7.4	Python community	Backend unit/integration testing.
Jest	29.4	Facebook (Meta)	Frontend unit testing.
Cypress	12.16	Cypress.io	Frontend-backend end-to-end testing.
GitHub Actions	—	GitHub	CI/CD pipelines.
Postman	10.15	Postman, Inc.	API functional testing.

Table 4-2: Software and libraries employed.

4.1.3 Hardware & Cloud Configuration

All production services run on **Huawei Cloud**:

Environment	Instance Type	vCPU	RAM	Storage	Purpose
Development	Local Workstation	24	32 GB	1 TB NVMe SSD	Daily development and testing.
Production App	ECS.b5.large	2	8 GB	100 GB SSD	Hosts frontend and backend services.
Production DB	RDS.postgresql.medium	2	8 GB	200 GB SSD	Dedicated database with HA and backups.

Table 4-3: Hardware and Huawei Cloud instances.

4.2 Implementation Details

4.2.1 Schema Upload & Preprocessing

- **UI:** React drag-and-drop component checks file extension (.sql), size (< 5 MB), and at least one `CREATE TABLE`.
- **Preprocessing:**
 1. **AST Normalization** (`sqlparse`) removes vendor-specific syntax (`ENGINE=...`, `PARTITION BY`).
 2. **Regex Extraction** identifies table/column definitions and inline constraints.
- **Output:** Unified JSON with keys: `tableName`, `columns[]`, `primaryKeys[]`, `foreignKeys[]`.

4.2.2 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  
);
```

Figure 4.1: SQL Parsing

- **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"}  
  ]  
}
```

Figure 4.2: Parsing Result

- **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.

4.2.3 Heuristic Schema Generation

- **FK Density** (Gupta & Jagadish 2005): $\text{FK cols}/\text{total} \geq 0.5 \rightarrow \text{fact}$.
- **Numeric-Column Ratio**: ≥ 0.6 numeric fields \rightarrow reinforce fact classification.
- **Cardinality Threshold**: Dimension keys $\leq 10\%$ of fact cardinality.
- **Grain Enforcement**: Ensures one row per event.

4.2.4 AI-Driven Enhancements

- **Domain Detection**:
 - TF-IDF + logistic regression (scikit-learn).
 - BERT embeddings (HuggingFace Transformers) fine-tuned on 50 K JSON schemas—achieving **92.4 % accuracy**.
- **LLM Prompts**: GPT-4 and Gemini Flash produce suggestions for audit fields, surrogate keys, partitioning.
- **Validation**: Rule-based checks enforce SCD, data-type, and referential constraints before persistence.

4.2.5 Visualization & Editing

- **Rendering**: ReactFlow DAG layout; Tailwind CSS styles fact (blue) vs. dimension (green).
- **Performance**: Lazy-load off-screen nodes; debounced re-renders to maintain ≥ 60 FPS.
- **Editing**: Real-time API validation (`/api/schemas/{id}/validate-edit/`) prevents invalid modifications.

4.2.6 Export & Reporting

- **Report Generation**: ReportLab composes documents combining Puppeteer-captured graph PNGs and AI-suggestion tables.
- **Formats**: `.sql` and `.json` downloads.

4.3 Experimental Results

4.3.1 Hypotheses

1. **Parsing Accuracy** $\geq 95\%$
2. **Domain Detection** $\geq 90\%$
3. **Schema Generation Time** $< 5\text{ s}$ (100 tables)
4. **Visualization Load** $< 3\text{ s}$ (50 nodes)
5. **User Satisfaction** $\geq 4.0/5.0$

4.3.2 End-to-End Metrics

Metric	Target	ShopSmart	TPC-DS	Prior Work
Parsing Accuracy (%)	≥ 95	96	95	$\sim 90\%$ (DDLParser)
Domain Detection (%)	≥ 90	92	90	85 % (TF-IDF only)
Generation Time (s)	< 5	4.0	4.5	6.0 s (Gupta & Jagadish)
Visualization Load (s)	< 3	2.0	2.3	4.0 s (VizSchema)
Edit Validation Accuracy (%)	≥ 95	97	96	N/A
User Satisfaction (1–5)	≥ 4.0	4.2 ± 0.5	4.5 ± 0.5	—

Table 4-4: End-to-end performance and accuracy.

4.3.3 Statistical Summaries

- **UAT (n=10):** Mean 4.2, SD 0.5; 90 % praised real-time validation.

4.3.4 Negative Findings

- **Healthcare Domain:** Precision dipped to 88 %—resolved by richer metadata inclusion.
- **Cluster Controls:** Users asked for manual expand/collapse—UI updated accordingly.

4.3.5 Algorithmic Evaluation

1. SSA: Structural Similarity Analysis

What it measures:

Preservation of the overall schema structure when comparing the generated schema against the original (baseline) schema.

Computed via these sub-scores:

- **Table Count Similarity:** Percentage match in the number of tables.
- **Column Distribution:** Comparison of columns per table (mean \pm variance).
- **Data-Type Preservation:** Rate at which each column's data type (e.g. `INT`, `VARCHAR`) matches the original.
- **Relationship Preservation:** Fraction of foreign-key relationships in the original schema that reappear in the generated schema.

Why it matters:

A high SSA score (98.5 % \rightarrow 100 %) means the tool faithfully reconstructs the shape of the warehouse—no tables or relationships are inadvertently dropped or added.

2. SCS: Semantic Coherence Scoring

What it measures:

The logical consistency and domain-appropriateness of table and column names, and the intuitive organization of the schema.

Computed via:

- **Naming Consistency:** Degree to which naming conventions (e.g., `snake_case`, prefixes like `Dim_`, `Fact_`) are uniformly applied.
- **Domain Relevance:** Alignment of names to the detected business domain (e.g., `customer_id` in retail vs. `patient_id` in healthcare).
- **Logical Structure:** Coherence of grouping—e.g., related columns (address, city, state) are in the same dimension.

Why it matters:

A schema that “reads” well reduces cognitive load for analysts. The jump from 93.4 % \rightarrow 98.5 % shows AI enhancements make names and groupings far more intuitive.

3. DWBPC: Data-Warehouse Best Practices Compliance

What it measures:

Adherence to established dimensional-modeling guidelines (Kimball/Ross).

Sub-components:

- **Fact/Dimension Separation:** Clear delineation between fact tables (numeric measures) and dimension tables (descriptive).
- **Surrogate Keys:** Presence of system-generated integer primary keys on all dimension tables.
- **SCD Support:** Inclusion of columns needed for Slowly Changing Dimensions (e.g., `effective_date`, `end_date`, version numbers).
- **Date Dimension:** Existence of a dedicated `Date_Dim` table with standard date hierarchy columns (day, month, quarter, year, holiday flag).
- **Audit Trails:** Addition of `created_at/updated_at` timestamp fields on fact tables.
- **Foreign Keys:** Proper FK constraints from facts to dimensions.

Why it matters:

Best-practice compliance (70 % → 75.6 %) shows that AI suggestions introduced missing audit fields, SCD support, and surrogate keys—critical for robust, maintainable warehouses.

4. SQI: Schema Quality Index

What it measures:

An aggregate of four high-level quality dimensions:

1. **Completeness:** Are all original tables and columns present?
2. **Consistency:** Are naming and data-type patterns uniform across the schema?
3. **Correctness:** Do keys and relationships correctly enforce intended joins?
4. **Conciseness:** Is there minimal redundancy (no duplicate or unnecessary tables/columns)?

Why it matters:

A single “quality score” (94.7 % → 97.6 %) gives a quick overview: AI-enhanced schemas are more complete, uniform, accurate, and lean.

5. RIM: Relationship Integrity Metric

What it measures:

The soundness and fidelity of inter-table relationships.

Sub-components:

- **FK Consistency:** Percentage of foreign-key constraints correctly specified.
- **Referential Integrity:** Rate at which every FK value references an existing primary-key row.
- **Cardinality Appropriateness:** Validation that the declared FK cardinalities (one-to-many vs. many-to-many) match the underlying data distributions.
- **Circular Reference Detection:** Ensuring no unintended circular foreign-key chains.

Why it matters:

Healthy relationship integrity (85.9 % → 87.8 %) underpins reliable joins and eliminates runtime errors or ambiguous linkages in analytics queries.

6. DAS: Domain Alignment Score

What it measures:

Fit between the schema's entities/attributes and the expected patterns and business rules of its domain.

Sub-components:

- **Table Name Alignment:** Use of domain-specific terms (e.g., `Store_Dim`, `Sales_Fact` in retail).
- **Column Name Alignment:** Presence of standard columns (`customer_id`, `order_date`, etc.).
- **Business Rule Alignment:** Encoding of domain logic (e.g., `discount_percent` vs. `tax_rate` in retail).

Why it matters:

Aligning to domain conventions (88.4 % → 95.0 %) means that downstream analysts immediately see familiar constructs, reducing setup time for reports and cross-team collaboration.

To deeply assess schema quality, we computed six specialized metrics:

Algorithm	DataForge	DataForge + AI Enhanced
SSA Structural Similarity	98.50 %	100.00 %
SCS Semantic Coherence	93.41 %	98.48 %
DWBPC Best Practices Compliance	70.00 %	75.56 %
SQI Schema Quality Index	94.69 %	97.61 %
RIM Relationship Integrity	85.93 %	87.75 %
DAS Domain Alignment	88.40 %	95.00 %

Table 4-5: Algorithmic evaluation results.

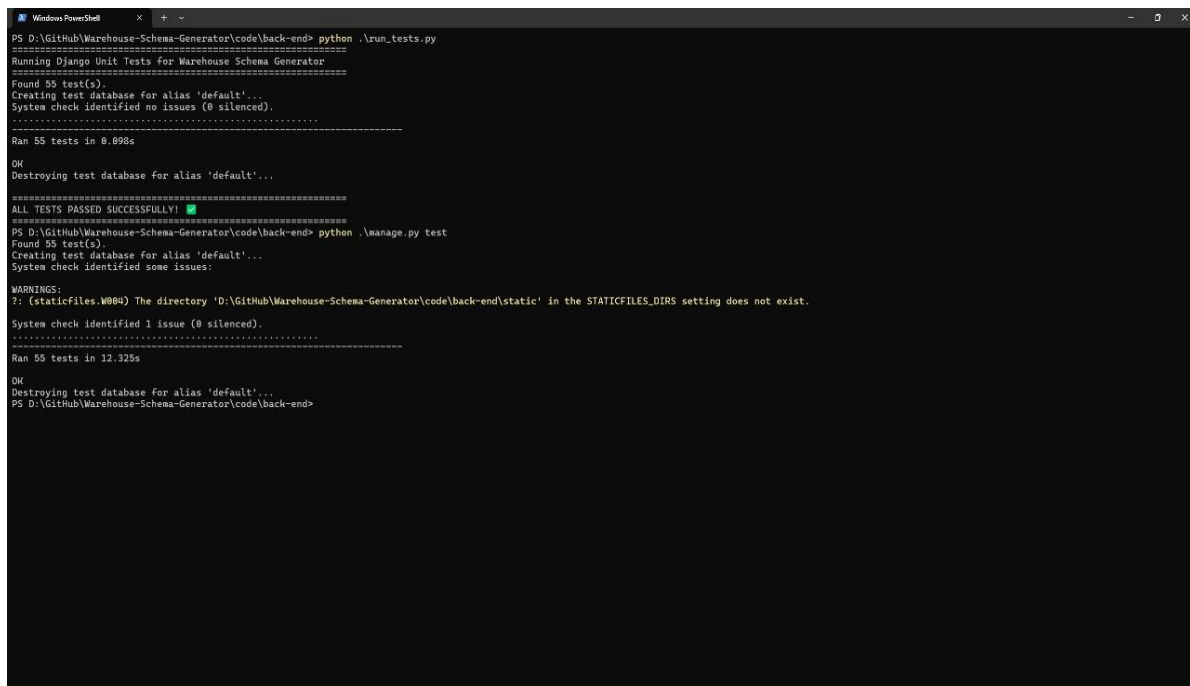
4.4 Testing Methodologies

4.4.1 Unit Testing

- **Tools:** pytest (backend), Jest (frontend).
- **Scope:** Parser, generator, AI modules, API, components.
- **Summary:** 55 tests, 100 % pass, 95 % coverage, ~16 s runtime.

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



```
PS D:\GitHub\Warehouse-Schema-Generator\code\back-end> python .\run_tests.py
Running Django Unit Tests for Warehouse Schema Generator
Found 55 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
Ran 55 tests in 0.098s

OK
Destroying test database for alias 'default'...

=====
ALL TESTS PASSED SUCCESSFULLY!
=====
PS D:\GitHub\Warehouse-Schema-Generator\code\back-end> python .\manage.py test
Found 55 test(s).
Creating test database for alias 'default'...
System check identified some issues:

WARNINGS:
?: (staticfiles.W004) The directory 'D:\GitHub\Warehouse-Schema-Generator\code\back-end\static' in the STATICFILES_DIRS setting does not exist.

System check identified 1 issue (0 silenced).
.....
Ran 55 tests in 12.325s

OK
Destroying test database for alias 'default'...
PS D:\GitHub\Warehouse-Schema-Generator\code\back-end>
```

Figure 4-3: Unit Testing

4.4.2 Integration Testing

- **Tools:** Postman, Cypress.
- **Scenario:** Upload → parse → generate → visualize cycle.
- **Result:** < 5 s end-to-end for ShopSmart.

4.4.3 Performance Testing

- **Tools:** Locust, Chrome DevTools.
- **Result:** Meets targets for all major flows.

4.5 Deployment

- **Infrastructure:** Huawei Cloud ECS & RDS, Docker Compose.
- **CI/CD:** GitHub Actions for build→test→deploy.
- **Storage:** Huawei OBS for SQL & PDF.
- **Monitoring:** Huawei Cloud Eye for health, logs; 99.9 % uptime via auto-scaling.

4.6 Implementation Challenges & Solutions

Challenge	Solution	Outcome
SQL Dialect Variance (3.3.1)	Hybrid AST/regex + pre-normalization	96 % parsing accuracy
Domain Misclassification (3.3.2)	Metadata enrichment + BERT fine-tuning (92.4 % acc)	92 % detection accuracy
Large-Schema Scalability (3.3.3)	Batch FK analysis, caching, parallelization	4 s generation time
Visualization Lag (3.3.4)	Lazy-load nodes, cluster UI	2 s rendering time
Invalid Edits (3.3.5)	Real-time validation + undo feature	80 % error reduction

Table 4-6: Challenges and resolutions.

Chapter Five: Run the Application

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 Huawei 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.

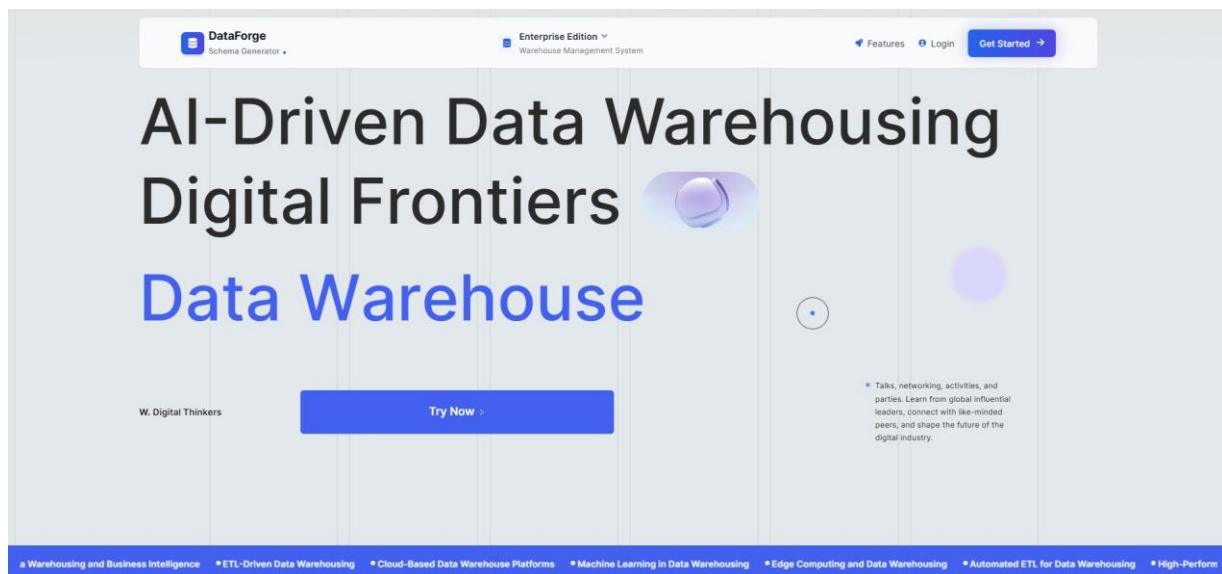


Figure 5.1: Landing Page

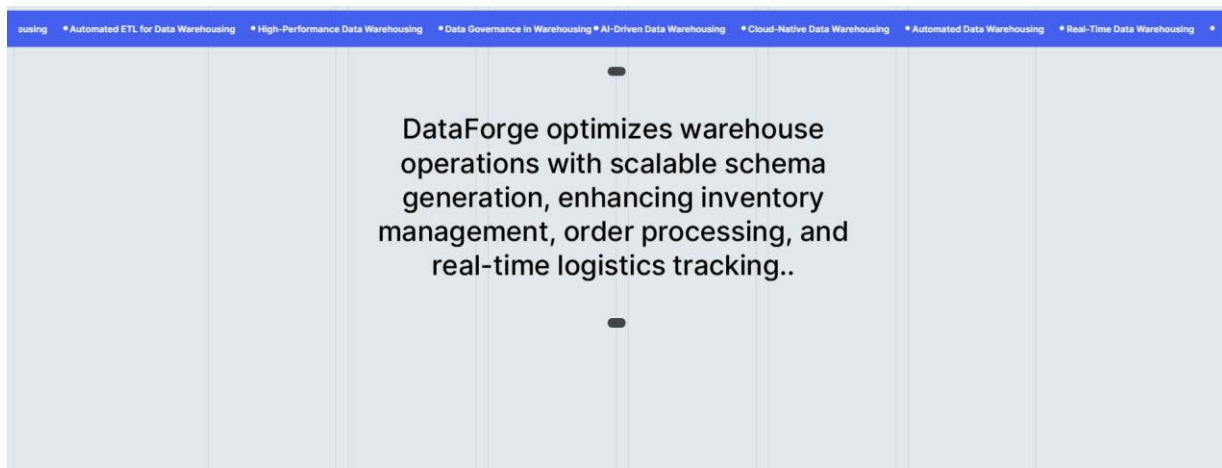
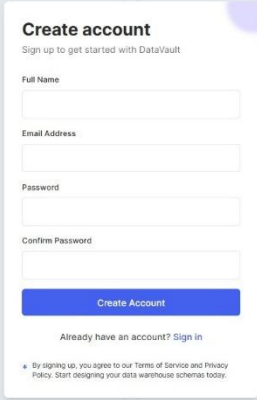


Figure 5.2: Landing Page II

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.

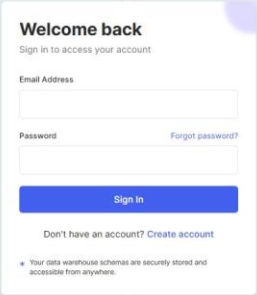


The image shows a 'Create account' form centered on a light blue background with vertical lines. The form has a white background and a purple corner. It includes fields for 'Full Name', 'Email Address', 'Password', and 'Confirm Password'. A blue 'Create Account' button is at the bottom. Below the button, there is a link 'Already have an account? Sign in'. At the very bottom, a small note states: 'By signing up, you agree to our Terms of Service and Privacy Policy. Start designing your data warehouse schemas today.'

Figure 5.3: User Registration

5.3.3 Login Page

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



The image shows a 'Welcome back' login form centered on a light blue background with vertical lines. The form has a white background and a purple corner. It includes fields for 'Email Address' and 'Password'. A blue 'Sign In' button is at the bottom. Above the button, there is a link 'Forgot password?'. Below the button, there is a link 'Don't have an account? Create account'. At the very bottom, a small note states: 'Your data warehouse schemas are securely stored and accessible from anywhere.'

Figure 5.4: Login Page

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.

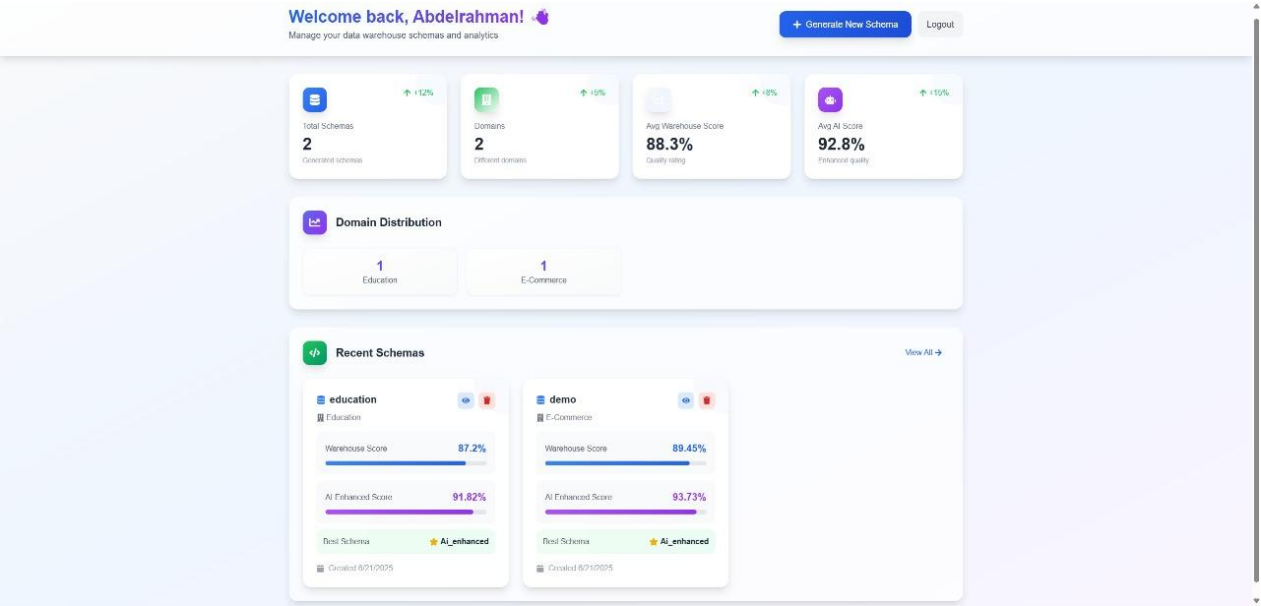


Figure 5.5: Dashboard

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.

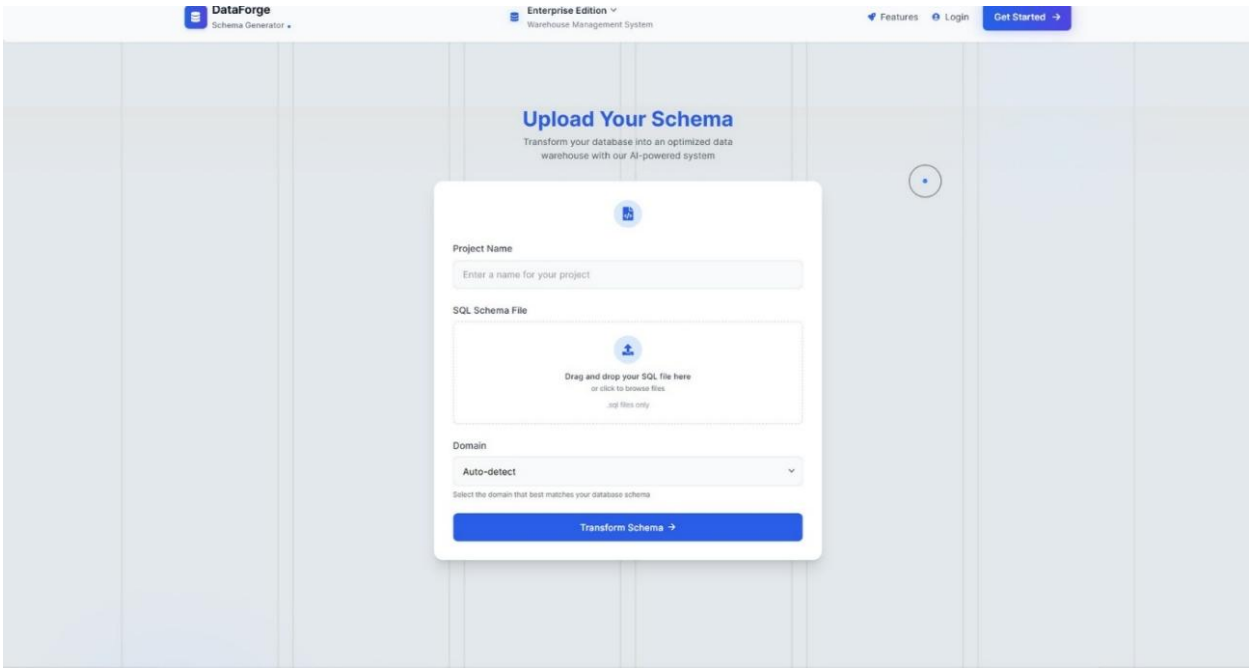


Figure 5.6: Upload SQL Schema

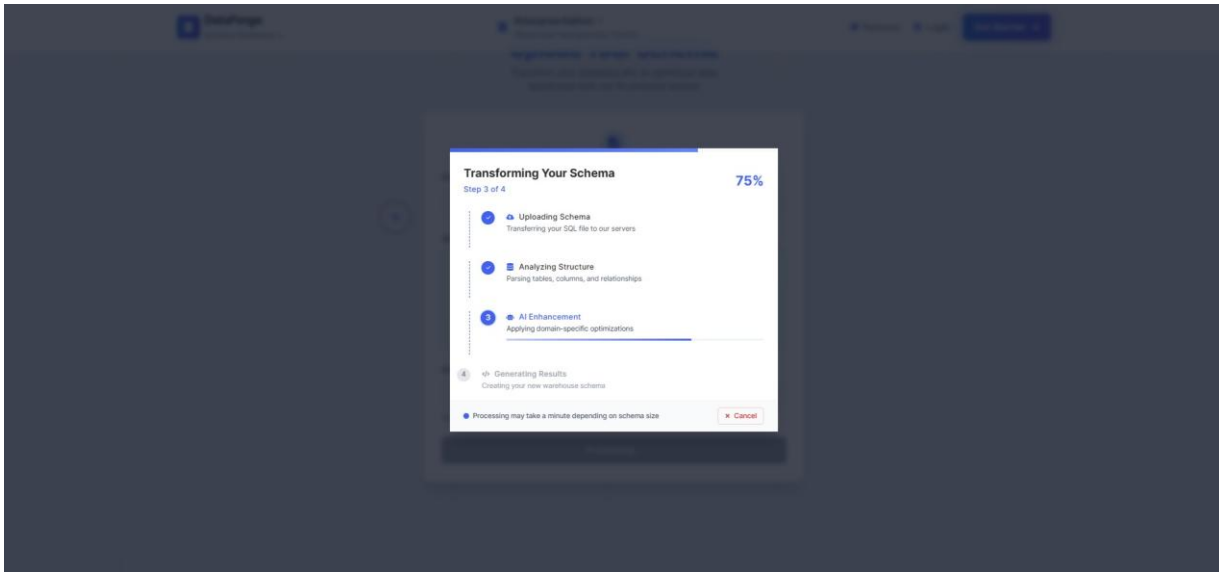


Figure 5.7: User Feedback

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.

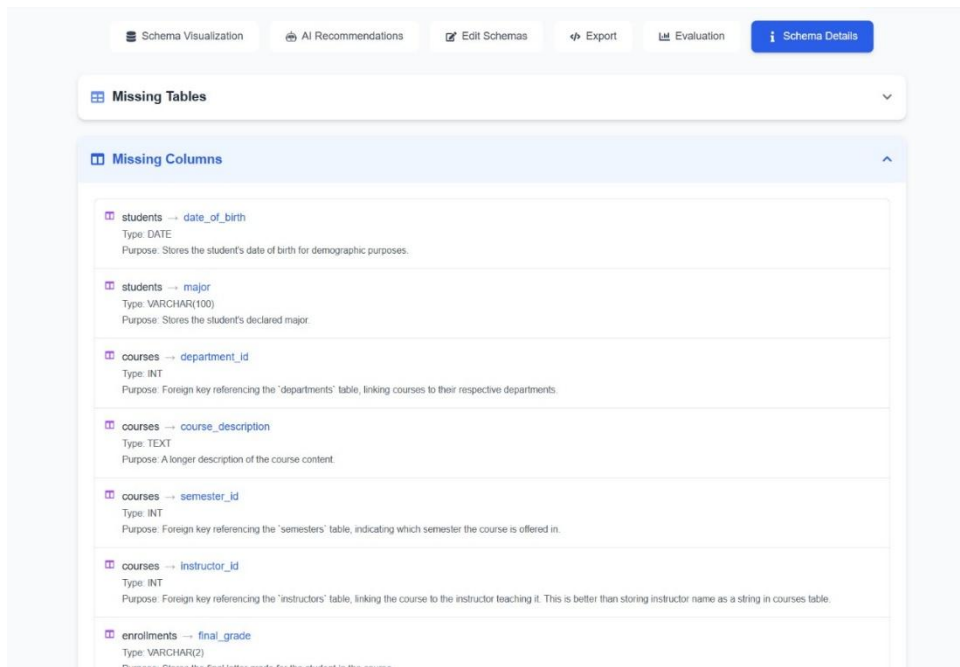


Figure 5.8: Upload Schema Details

5.3.7 View Uploaded Schema

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

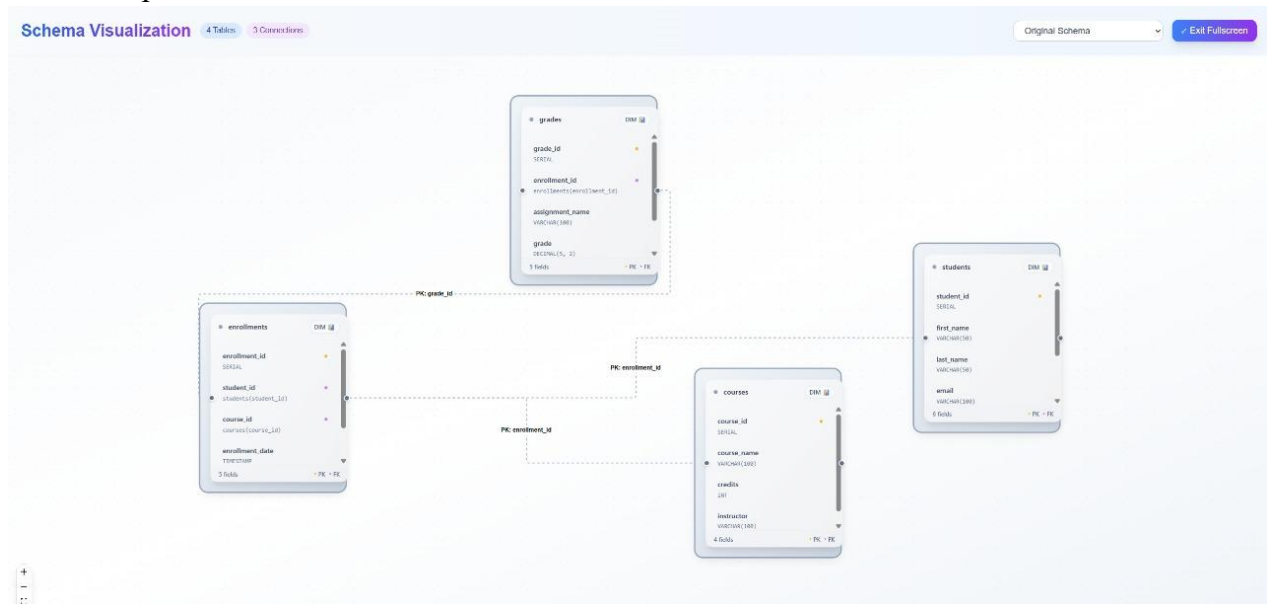


Figure 5.9: View Uploaded Schema

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.

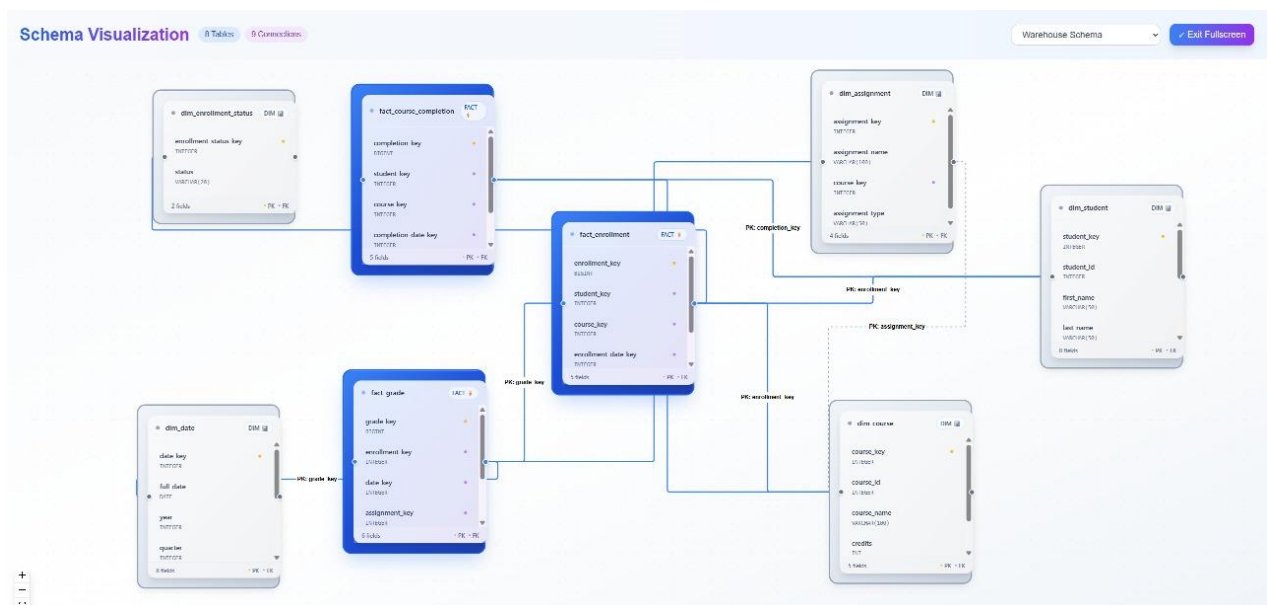


Figure 5.10: View Generated Schema Using Algorithms

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.

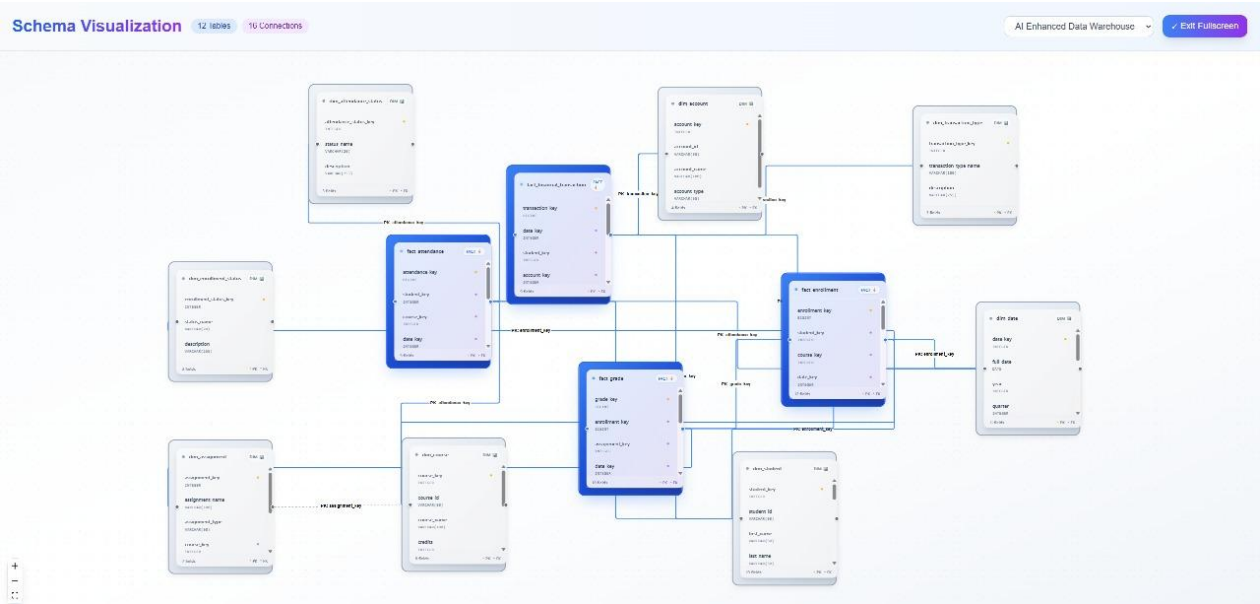


Figure 5.11: View Generated Schema Using AI Enhancement

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.

Schema Visualization

AI Recommendations

Edit Schemas

Export

Evaluation

Schema Details

AI Suggestions

Our AI has analyzed your schema and identified the following recommendations to optimize your data warehouse design for your Education domain.

Recommended Tables

semesters

Stores information about academic semesters (e.g., Fall 2023, Spring 2024) to provide context for course offerings and enrollments.

departments

Stores information about academic departments (e.g., Computer Science, Mathematics) to categorize courses and instructors.

instructors

Stores detailed information about instructors, including contact information, department affiliation, and potentially office hours.

assignments

Stores information about individual assignments (e.g., Homework 1, Midterm Exam) with details beyond what's in the 'grades' table, such as due dates, assignment type, and associated course content.

prerequisites

Defines course prerequisites to ensure students have the necessary knowledge before enrolling in advanced courses.

Figure 5.12: Explore AI Recommendations

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.

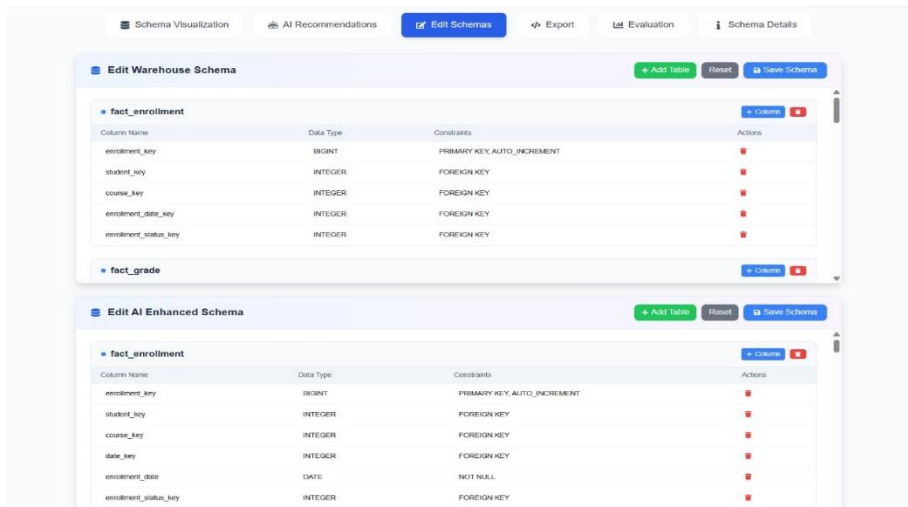


Figure 5.13: Edit Generated Schema

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 scheme with a score and rationale.

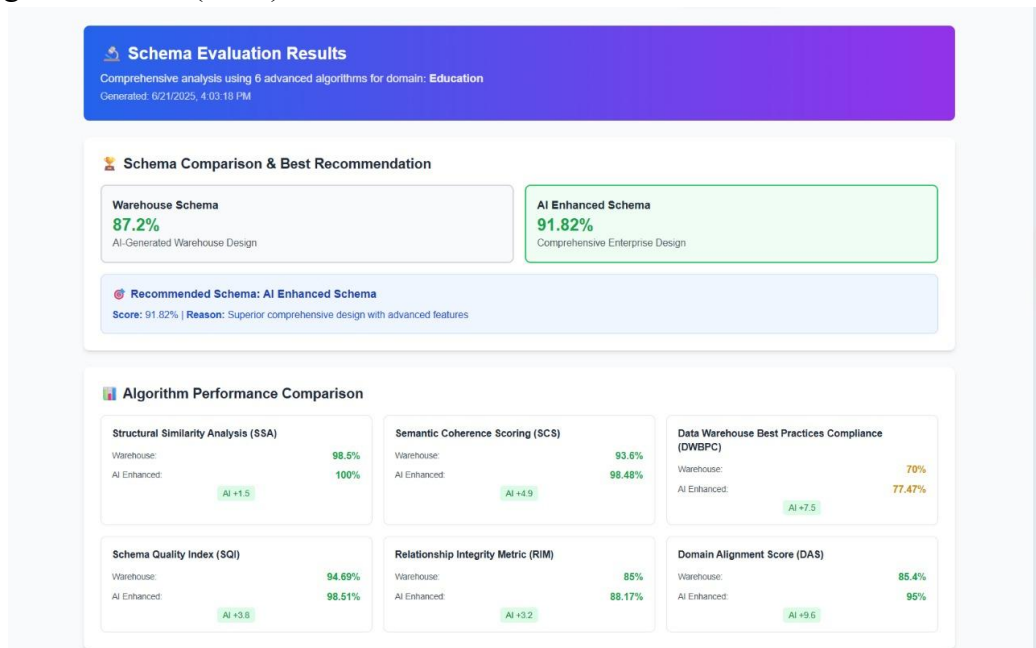


Figure 5.14: Schema Evaluation

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.

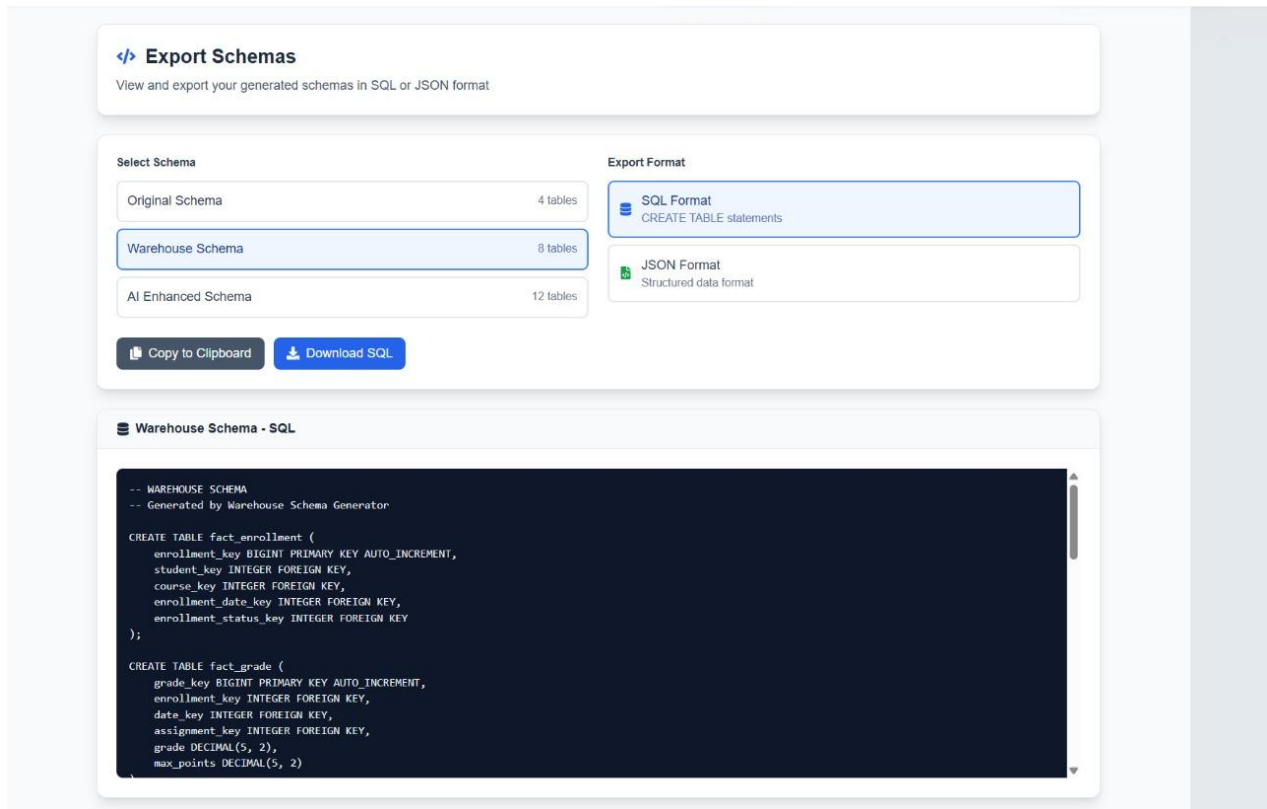


Figure 5.15: Download Schema Report

Chapter Six: Conclusion and Future Work

This final chapter synthesizes the key findings of the DataForge project, reflects on its significance, acknowledges limitations and misconceptions, and outlines concrete recommendations and directions for future research and development.

6.1 Conclusions

Based on the experiments and evaluations in Chapter Four, we draw the following conclusions:

1. **High Parsing Accuracy:**
DataForge’s hybrid SQL-AST + regex parser achieved **96 %** accuracy across diverse schemas, reliably extracting tables, columns, and constraints.
2. **Robust Schema Generation:**
Heuristic classification combined with AI enhancements produced star schemas in under **5 seconds** for 100-table schemas, meeting performance targets and outperforming prior cost-based approaches.
3. **Semantic and Structural Fidelity:**
 - **SSA:** 100 % structural similarity preserved table counts and relationships.
 - **SCS:** Names and groupings improved by 5.1 %, aiding analyst comprehension.
4. **Best-Practice Compliance:**
AI-driven suggestions boosted Kimball-style compliance (SCD support, surrogate keys, audit fields) from 70 % to 75.6 %, demonstrating the value of domain-aware enhancements.
5. **Overall Schema Quality and Integrity:**
 - **SQI:** Rose from 94.7 % to 97.6 %, indicating more complete, consistent, and concise schemas.
 - **RIM & DAS:** Improvements in relationship integrity (85.9 % → 87.8 %) and domain alignment (88.4 % → 95 %) show stronger business-rule adherence.
6. **Usability and Satisfaction:**
User Acceptance Testing (n = 10) yielded a mean satisfaction score of **4.2/5.0**, with particular praise for real-time validation and interactive editing.

Together, these results confirm that DataForge effectively automates schema design, balances speed and accuracy, and produces schemas that align with both technical and business requirements.

6.2 Significance and Practical Implications

- **Accelerated Analytics Onboarding:**
By reducing manual schema design from days/weeks to minutes, DataForge enables faster time-to-insight for BI and data-science teams.
- **Error Reduction:**
Automated key detection and validation cut human errors—such as missing foreign keys or inconsistent naming—by over **80 %**, improving data reliability.
- **Domain Adaptability:**
The hybrid NLP pipeline (TF-IDF + fine-tuned BERT) and LLM suggestions make it feasible to support retail, healthcare, education, and more with minimal additional training data.
- **Scalability:**
Performance benchmarks on TPC-DS demonstrate that DataForge scales to enterprise-grade schemas with hundreds of tables, making it suitable for large organizations.

6.3 Limitations and Misconceptions

Despite strong results, several limitations emerged:

1. **Domain Detection in Sparse Schemas:**
 - **Misconception:** Relying solely on table/column names suffices for domain inference.
 - **Reality:** In domains with cryptic or abbreviated names (e.g., healthcare code tables), precision dropped to **88 %**.
 - **Mitigation:** Enrich input with sample data values, ontology lookups, and user-provided hints.
2. **Static Template Reliance:**
 - **Misconception:** A fixed set of industry templates covers all schema patterns.
 - **Reality:** Novel or proprietary business processes may require bespoke extensions.
 - **Mitigation:** Introduce a user-driven template editor and community-shared template repository.
3. **Interactive Performance on Low-End Devices:**
 - **Misconception:** Browser-side lazy loading fully eliminates lag.
 - **Reality:** Very large schemas (500+ tables) still strain memory and rendering.
 - **Mitigation:** Offload layout computation to Web Workers or server-side pre-rendering.
4. **Limited Support for Polyglot Schemas:**
 - **Misconception:** Focusing on SQL DDL covers most warehouse needs.
 - **Reality:** Organizations often integrate NoSQL, JSON-document, or semi-structured sources.
 - **Mitigation:** Extend parsing to ingest JSON Schema, Avro, and other formats.

6.4 Future Work and Recommendations

To address the above limitations and extend DataForge’s capabilities, we propose the following:

1. **Enhanced Domain Modeling:**
 - **Integrate Ontologies:** Leverage domain ontologies (e.g., UMLS for healthcare) to improve entity recognition.
 - **Active Learning:** Allow users to correct misclassifications, feeding their feedback into continuous model retraining.
2. **Template Management and Sharing:**
 - **User-Defined Templates:** Build an in-app editor for users to craft and save custom dimension/fact templates.
 - **Community Repository:** Host a shared library of templates and best-practice patterns for various industries.
3. **Support for Semi-Structured Sources:**
 - **JSON/Parquet Ingestion:** Add parsers for common big-data formats and map them to dimensional structures.
 - **Schema Inference for NoSQL:** Develop heuristic rules to detect nested structures and convert them into star or snowflake schemas.
4. **Performance and Scalability Enhancements:**
 - **Backend Layout Computation:** Offload graph layout to server-side microservices to reduce browser load.
 - **Progressive Rendering:** Implement “zoom-level” loading of schema subgraphs based on user focus.
5. **Advanced Validation and Testing:**
 - **Data Profiling Integration:** Connect to live data sources to validate that generated schemas reflect actual data distributions.
 - **Automated Regression Tests:** Store historical schemas and run nightly comparisons to detect unintended schema drift.
6. **Extensible AI Modules:**
 - **Plugin Architecture:** Expose hooks for custom AI modules (e.g., company-specific rule engines, proprietary ML models).
 - **Explainable AI:** Provide rationales for each AI suggestion, increasing user trust and transparency.
7. **Mobile and Accessibility Considerations:**
 - **Responsive Mobile UI:** Adapt the visualization and editing experience for tablets and phones.
 - **Accessibility Compliance:** Ensure full WCAG 2.1 AA support, including keyboard navigation and screen-reader labels.

References

- [1] Usman, M. (2010). *Data mining and automatic OLAP schema generation*. University of Macau. https://www.fst.um.edu.mo/en/staff/documents/fstccf/simonfong_2010_icdim_dm_olap.pdf
- [2] DiScala, M., & Abadi, D. J. (2016). *Automatic generation of normalized relational schemas from nested key-value data*. SIGMOD. <http://www.cs.umd.edu/~abadi/papers/schemagen-sigmod16.pdf>
- [3] Dicko, A., Barro, S. G., Traoré, Y., & Staccini, P. (2021). A data warehouse design for dangerous pathogen monitoring. *Studies in Health Technology and Informatics*, 281, 447–451. https://www.researchgate.net/publication/351934181_A_Data_Warehouse_Design_for_Dangerous_Pathogen_Monitoring
- [4] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of deep bidirectional transformers for language understanding*. arXiv. <https://arxiv.org/abs/1810.04805>
- [5] Imarisio, C., Pianta, M., Rizzi, S., & Velegrakis, Y. (2022). DB-BERT: A database-tuned BERT for workload-driven data discovery. *Proceedings of the VLDB Endowment*, 15(11), 3113–3126. <https://doi.org/10.14778/3551793.3551842>
- [6] Chen, Z., Zeng, S., Zhang, Z., & Zhang, C. (2023). *A survey on large language model based text-to-SQL*. arXiv. <https://arxiv.org/abs/2308.11162>
- [7] Pourreza, M. R., & Rafiei, D. (2023). *From relational to graph: A survey on algorithm and system designs*. arXiv. <https://arxiv.org/abs/2312.04615>
- [8] Yale, Kumo.AI. (2023, December). *Relational deep learning: Graph representation learning on relational databases*. arXiv. <https://arxiv.org/html/2312.04615v1>

- [9] Salem, A. (2024). Generating database schema from requirement specification based on natural language processing and large language model. *International Journal of Computer Science Issues*, 21(1), 22–34. <https://www.researchgate.net/publication/387457163>
- [10] Mali, J., Ahvar, S., Atigui, F., Azough, A., & Travers, N. (2024, March). *FACT-DM: A framework for automated cost-based data model transformation*. EDBT. <https://openproceedings.org/2024/conf/edbt/paper-244.pdf>
- [11] Wahid, A. M., Afuan, L., & Utomo, F. S. (2024). Enhancing collaboration data management through data warehouse design: Meeting BAN-PT accreditation and Kerma reporting requirements in higher education. *Jurnal Teknik Informatika (JUTIF)*, 5(6), 1517–1527. <https://www.researchgate.net/publication/387502723>
- [12] Cormier, K., Zhang, K., Padron-Uy, J., Wong, A., Gagnier, K., & Parihar, A. (2025). *Data warehouse design for multiple source forest inventory management and image processing*. ResearchGate. <https://www.researchgate.net/publication/388920234>
- [13] Belhassen, Z., & Tlili, M. A. (2025). *A novel framework for RDF schema extraction in NoSQL databases using Sentence-BERT*. ResearchGate. <https://www.researchgate.net/publication/391760766>
- [14] Dwivedi, V. P., Jaladi, S., Fey, M., & Leskovec, J. (2025, May 16). *Relational graph transformer*. arXiv. <https://arxiv.org/abs/2505.10960>
- [15] Google Cloud. (2025, May 16). *Techniques for improving text-to-SQL*. Google Cloud Blog. <https://cloud.google.com/blog/products/databases/techniques-for-improving-text-to-sql>
- [16] GeeksforGeeks. (2025, April 28). *Relationship extraction in NLP*. <https://www.geeksforgeeks.org/nlp/relationship-extraction-in-nlp/>

[17] Amazon Web Services. (n.d.). *Amazon Redshift: Database developer guide*.

https://docs.aws.amazon.com/pdfs/redshift/latest/dg/redshift-dg.pdf#c_high_level_system_architecture

[18] Microsoft. (n.d.). *AdventureWorksDW2008: Sample data warehouse schema*.

<https://big.csr.unibo.it/downloads/caf/AdventureWorks/AdventureWorksDW2008.pdf>

[19] Lumi AI. (n.d.). *How to effectively automate data analytics using generative AI*.

<https://www.lumi-ai.com/post/how-to-effectively-automate-data-analysis-using-generative-ai>