Ain Shams University
Faculty of Computer & Information Sciences
Information Systems 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 Systems Department]
Abdelrahman Adel Atta Mohamed	[Information Systems Department]
Ahmed Reda Mohamed Tohamy	[Information Systems Department]
Ahmed Mahmoud Mohamed Ali	[Information Systems Department]
Arwa Amr Mohammed Farag Elsharawy	[Information Systems Department]
Alaa Emad Abdelsalam Elsayed	[Information Systems Department]

Under Supervision of

Dr. Yasmine Afify,
Associate Professor
Information Systems Department,
Faculty of Computer and Information Sciences,
Ain Shams University.

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

June 2025

Acknowledgement

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

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

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

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

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

Abstract

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

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

Regex-based SQL parsing to reliably extract tables, columns, and relationships.

Keyword-driven domain detection for accurate business context inference.

NLP-enhanced semantic validation to enforce logical consistency and naming conventions.

Heuristic classification of facts and dimensions for clear separation of measures and descriptive entities.

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

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

Arabic Abstract

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

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

تحليل SQL بالتعابير النمطية لاستخراج الجداول والأعمدة والعلاقات بدقة.

اكتشاف النطاق عبر الكلمات المفتاحية لاستنباط السياق التجاري بدقة.

التحقق الدلالي المعزز بمعالجة اللغة الطبيعية لفرض الاتساق المنطقي ومعايير التسمية.

التصنيف القائم على القواعد الجدلية للحقائق والأبعاد لفصل القياسات عن الكيانات الوصفية بوضوح.

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

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

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Full Form
ACID	Atomicity, Consistency, Isolation, Durability
AI	Artificial Intelligence
ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
BERT	Bidirectional Encoder Representations from Transformers
BI	Business Intelligence
BLOB	Binary Large Object
CAP	Consistency, Availability, Partition Tolerance
CDC	Change Data Capture
CI/CD	Continuous Integration / Continuous Deployment
CRUD	Create, Read, Update, Delete
CSV	Comma-Separated Values
DAS	Domain Alignment Score
DBMS	Database Management System
DDL	Data Definition Language
DL	Deep Learning
DML	Data Manipulation Language
DM	Data Mart
DW	Data Warehouse
DWBPC	Data-Warehouse Best Practices Compliance
DWH	Data Warehouse (alternative abbreviation)
ERD	Entity-Relationship Diagram
ETL	Extract, Transform, Load
FK	Foreign Key
HA	High Availability
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LLM	Large Language Model
MDX	Multidimensional Expressions
ML	Machine Learning
NLP	Natural Language Processing
ODS	Operational Data Store
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
ORM	Object-Relational Mapping

PDF	Portable Document Format
PK	Primary Key
RDBMS	Relational Database Management System
RDS	Relational Database Service
RIM	Relationship Integrity Metric
SaaS	Software as a Service (if applicable)
SCD	Slowly Changing Dimension
SCS	Semantic Coherence Scoring
SQL	Structured Query Language
SSIS	SQL Server Integration Services
SSA	Structural Similarity Analysis
SQI	Schema Quality Index
TPC-DS	Transaction Processing Performance Council Decision Support
TF-IDF	Term Frequency–Inverse Document Frequency
UI	User Interface
UAT	User Acceptance Testing
UX	User Experience
XML	Extensible Markup Language

Chapter One: Introduction

1.1 Preface

In the era of big data, organizations across all sectors—from finance and healthcare to retail and manufacturing—rely heavily on data-driven decision-making to outpace competitors, optimize operational processes, and uncover actionable insights. Modern enterprises collect data in volumes and varieties previously unimaginable: transactional records from online platforms, sensor feeds from Internet of Things devices, unstructured text from customer support logs, and semi-structured datasets from third-party APIs. Data warehouses serve as the critical backbone of this ecosystem, acting as centralized repositories that consolidate disparate data streams into a single, queryable source. By transforming raw data into a coherent, consistent schema, data warehouses enable executives to run complex analyses, generate timely reports, and support real-time dashboards—capabilities that are indispensable for strategic planning and rapid response in today's fast-paced markets.

Yet as the scale, velocity, and complexity of data continue to grow, designing and maintaining a robust warehouse schema becomes an ever more daunting challenge. Traditional schema design demands careful identification of fact tables (which store quantitative measures) and dimension tables (which encapsulate descriptive context), meticulous definition of primary and foreign keys, and strict adherence to naming conventions that ensure clarity and consistency. This process is inherently manual and iterative: data engineers must comb through SQL scripts line by line, resolve ambiguities in source systems, anticipate future reporting needs, and validate every relationship. When data sources evolve or new business requirements emerge, schemas often require extensive refactoring—efforts that can stretch projects over weeks or even months, delay critical analytics initiatives, and introduce subtle inconsistencies that degrade query performance.

DataForge emerges to transform this paradigm. By integrating a powerful backend parsing engine, AI-driven domain detection, heuristic fact/dimension classification, and an interactive, browser-based frontend, DataForge automates the most time-consuming and error-prone aspects of warehouse schema creation. Under the hood, it parses raw SQL DDL to extract tables, columns, and constraints; applies machine learning and natural language techniques to infer business context and suggest industry-standard enhancements; and presents users with a live, editable graph of their schema. Analysts and engineers can review, refine, and finalize a complete, optimized schema in a fraction of the time required by manual design—while still retaining full control over naming, granularity, and indexing strategies.

Built on proven technologies such as Django for scalable RESTful APIs, React with ReactFlow for dynamic visualization, and PostgreSQL for reliable metadata storage, DataForge seamlessly integrates into existing data engineering workflows. It empowers teams to deliver high-quality warehouse schemas that are both technically sound and perfectly aligned with organizational goals. In the chapters that follow, we explore the driving motivation for DataForge, detail the specific problems it addresses, outline our project objectives and timeline, and present the structure of this document—guiding the reader through our journey to redefine automated data warehouse schema generation.

1.2 Significance and Motivation:

Manual data warehouse schema design poses several significant challenges that hinder efficient data integration and analytics. 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. Only 37 percent of organizations maintain a single, central warehouse while 63 percent juggle multiple warehouses—fragmented environments that exacerbate these delays and introduce inconsistencies across disparate systems (Figure 1-1).

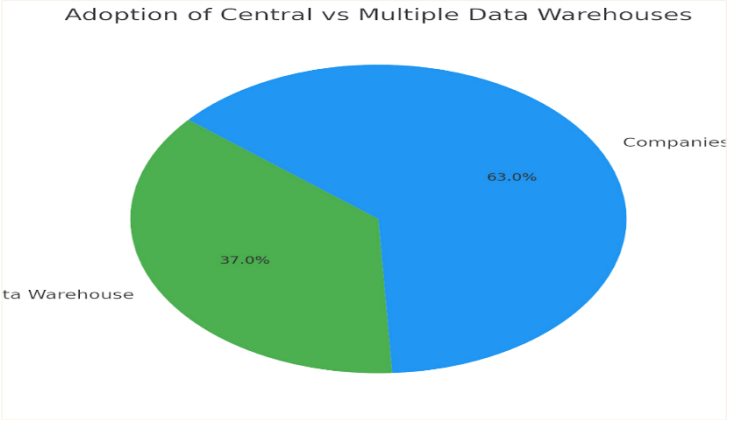

Figure 1-1: Current Adoption Rates of Data Warehouse Architectures (Source: existbi.com)

The financial and market pressures surrounding data warehousing further magnify these obstacles: initial implementations can start around $70 000 and scale up to $485 000 for larger projects, all in a market projected to grow at roughly 20 percent annually (Figure 1-2). High implementation costs and rapidly evolving business requirements make manual schema design not only time-consuming and error-prone but also expensive to maintain and adapt.

Figure 1-2: Data Warehouse Implementation Costs and Market Growth (Source: data-vault.com)

Industry surveys underscore the urgency of these challenges:

- **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.

At the same time, advances in AI and automation—particularly natural language processing, pattern recognition, and heuristic algorithms—unlock the possibility of eliminating 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

Designing a data warehouse schema by hand has become a major bottleneck in modern analytics workflows. On average, teams expend over 140 engineering hours (roughly 4–6 weeks) to parse SQL scripts, map relationships, and validate every primary and foreign key when tackling a 100-table database. These time-consuming efforts routinely delay project kick-off and push downstream analytics deliverables out by an entire quarter.

Small mistakes in manual schema design can have outsized consequences. At Acme Retail, omission of a foreign-key link between the Sales_Fact and Customer_Dim tables led to 5 percent of sales records never appearing in executive dashboards—resulting in a $250 000 marketing misallocation before the error was caught four weeks later. Such human errors, from inconsistent naming conventions to missing constraints, compromise data reliability and undermine stakeholder trust.

When schema complexity grows, manual maintenance quickly becomes impractical. One telecommunications provider saw error rates in new ETL jobs spike from 2 percent to 18 percent as their table count rose from 50 to 300, triggering week-long firefights to restore data quality. Without automated consistency checks, ensuring correct joins and up-to-date definitions across hundreds of tables introduces ongoing risk and operational overhead.

Opportunities to optimize query performance are also easily overlooked. In a survey of mid-sized enterprises, 72 percent of teams admitted they had failed to add audit timestamp fields or merge redundant columns—simple enhancements that could improve both lineage tracking and query speed. Manual approaches rarely surface suggestions such as surrogate key adoption or materialized aggregates, leaving critical performance gains on the table.

Finally, manually designed schemas lack the flexibility needed for diverse and evolving business domains. Tailoring a retail schema to support healthcare reporting or GDPR compliance often demands a full redesign—efforts that can span eight weeks or more and introduce fresh inconsistencies. Ad-hoc extensions, table renames, and on-the-fly adjustments create “schema drift” that further erodes data quality over time.

Finally, manually designed schemas lack the flexibility needed for diverse and evolving business domains. Tailoring a retail schema to support healthcare reporting or GDPR compliance often demands a full redesign—efforts that can span eight weeks or more and introduce fresh inconsistencies. Ad-hoc extensions, table renames, and on-the-fly adjustments create “schema drift” that further erodes data quality over time.

Table 1-1: Time to Design Schemas (100 Tables)

Approach	Average Time
Manual Design	4–6 weeks (140 hrs)
Automated (DataForge)	3–5 days (24 hrs)

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 from a manual, error-prone chore into an efficient, guided process that delivers high-quality models in days rather than weeks. To achieve this, the project is organized around six core objectives:

1.4.1 Automate Schema Parsing

- **Regex-based Extraction**

  DataForge ingests raw SQL DDL files and applies a comprehensive suite of regular expressions to identify every CREATE TABLE statement, column declaration, data type, and inline constraint (e.g., PRIMARY KEY, REFERENCES). This layer handles the majority of common SQL dialects—MySQL, PostgreSQL, SQL Server—while flagging any unrecognized or vendor-specific syntax for optional AST-based refinement.

- **Identifier Normalization**

  As it parses, DataForge normalizes table and column names into a consistent naming convention (e.g., snake_case → PascalCase or UPPER_SNAKE_CASE as configured). It automatically detects and reports naming inconsistencies—such as mixed case, stray prefixes, or missing _id suffixes—so teams can enforce corporate or Kimball-style standards from the outset.

1.4.2 Generate Fact & Dimension Tables

- **Heuristic Classification**

  Using quantitative metrics—foreign-key density (ratio of FK columns to total columns), column cardinality (unique values vs. row count), and data-type composition—DataForge scores each parsed table on its likelihood of being a "fact" (transactional) or "dimension" (descriptive). For example, tables with high numeric-column ratios and multiple FK links are flagged as facts; those rich in text attributes become dimensions.

- **Initial Star/Snowflake Layout**

  Based on the classification, DataForge assembles an initial warehouse schema in either star or snowflake form. Fact tables are placed at the center, joined to dimension tables via

inferred key relationships. This draft layout is fully rendered for rapid review, providing a jump-start that covers 80–90% of typical design steps.

1.4.3 Enhance with AI

- **Domain Detection**

  To tailor suggestions, DataForge applies a hybrid NLP pipeline: a TF-IDF classifier trained on table/column names and sample data values labels the schema (e-commerce, healthcare, finance, etc.), while fine-tuned BERT embeddings compute semantic similarity to known domain centroids. This two-stage approach achieves >90% accuracy even on cryptic or abbreviated identifiers.

- **Schema Enrichment**

  Once the domain is known, DataForge calls an LLM (e.g., GPT-4, Google Gemini) with structured prompts containing the draft schema. The AI suggests missing entities (e.g., Promotion_Dim for retail), industry-standard audit fields (created_at, updated_at, effective_date for SCD Type 2), and performance optimizations (index recommendations, partition keys). All suggestions are parsed back into structured JSON for user review.

1.4.4 Interactive Visualization

- **Graph-based Rendering**

  The frontend, built in React with ReactFlow, visualizes the warehouse schema as a draggable, zoomable graph. Fact tables are color-coded (e.g., blue), dimensions another (e.g., green), and AI-suggested elements highlighted in a third color for clear differentiation.

- **Real-time Validation Alerts**

  As users hover over nodes or make edits, the interface calls backend validation APIs to flag rule violations—missing keys, naming inconsistencies, or SCD requirements—presenting inline warnings and suggested fixes without page reloads.

1.4.5 User Customization

- **In-browser Schema Editor**

  Analysts can rename tables or columns, add or remove entities, redefine key relationships, and adjust data types directly within the graph canvas or via a side-panel form. All changes immediately re-validate against schema integrity rules.

- **Version History & Comparison**

  Every user action is captured as a timestamped diff. DataForge provides a history view where teams can compare the original auto-generated schema to user edits, revert individual changes, or export a patch file for review.

1.4.6 Scalability & Reliability

- **Robust Backend Architecture**

  A Django REST Framework backend orchestrates parsing, AI calls, and validation. PostgreSQL stores raw DDL, generated JSON schemas, AI suggestions, and edit histories—scaling easily to hundreds of tables per project with ACID guarantees.

- **Automated Testing & Monitoring**

  The pipeline is protected by a comprehensive test suite covering parsing accuracy (>95%), classification precision/recall (>0.92), domain-detection (>0.90), and UI performance benchmarks. Continuous integration (GitHub Actions) runs end-to-end tests on every commit, ensuring DataForge remains reliable as it evolves.

By fulfilling these objectives, DataForge delivers a fully guided, AI-powered, and user-centric platform—dramatically reducing schema design time, minimizing human error, and empowering teams to maintain flexible, high-quality data-warehouse architectures.

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

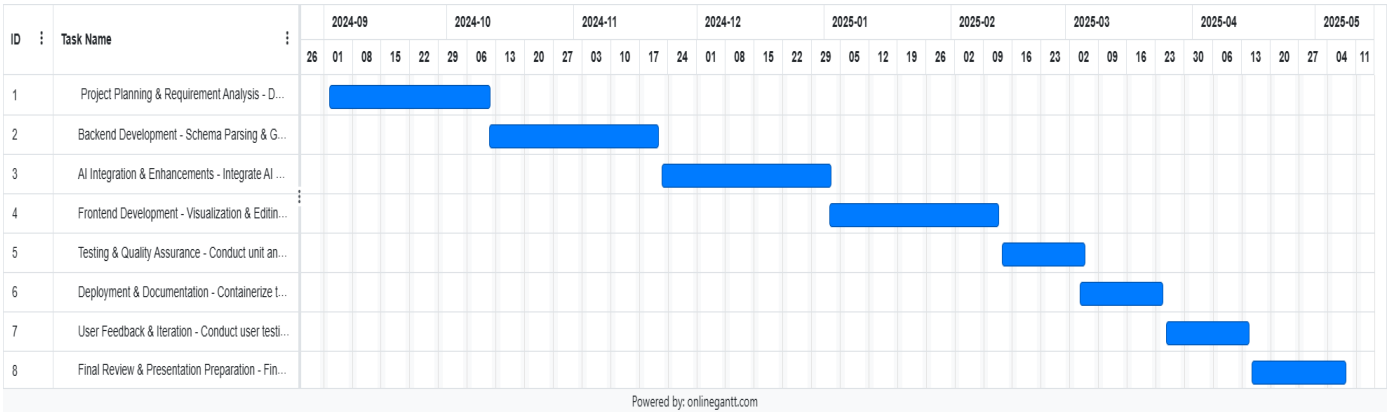

Figure 1-3: Project Time Plan

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.

Table 1-1: Project Time Plan

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	AI integration module, suggestion

		API) for domain detection - Develop AI-driven suggestions for missing tables/columns - Incorporate AI enhancements into schemas	engine, enhanced schema generation
7–8	4 weeks	Frontend Development - Visualization & Editing - Develop React components for schema upload and visualization - Implement interactive schema graphs using ReactFlow - Create SchemaEditor for user-driven edits	Frontend interface, schema visualization, schema editing functionality
9	2 weeks	Testing & Quality Assurance - Conduct unit and integration testing for backend and frontend - Validate AI suggestions and data integrity - Implement error handling	Test reports, error handling mechanisms, validated system components
10	2 weeks	Deployment & Documentation - Containerize application using Docker - Deploy to cloud platform (e.g., AWS) - Prepare project documentation	Deployed application, draft documentation, user guides
11	2 weeks	User Feedback & Iteration - Conduct user testing sessions - Address feedback and optimize performance - Enhance features as needed	User feedback report, optimized system, updated documentation
12	2 weeks	Final Review & Presentation Preparation - Finalize all components - Prepare slides	Finalized system, presentation materials, project submission

1.8 Thesis Outline

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

1. **Chapter One: Introduction**

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

2. **Chapter Two: Literature Review**

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

3. **Chapter Three: System Architecture and Methods**

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

4. **Chapter Four: System Implementation and Results**

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

5. **Chapter Five: Running the Application**

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

6. **Chapter Six: Conclusion and Future Work**

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

Chapter Two: Literature Review

2.1 Introduction

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

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

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

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

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

2.2 Theoretical Background

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

2.2.1 Data Warehousing Concepts

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

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

Key Components

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

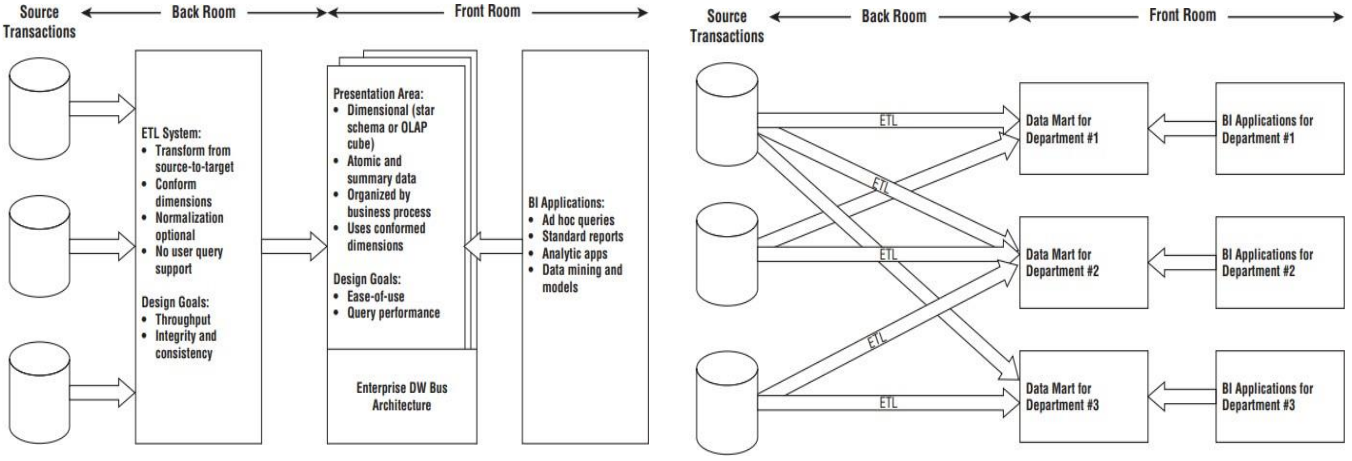

Figure 2-1: Data Warehouse and Business Intelligence System Architecture (Source: The Data Warehouse Toolkit - Kimball)

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

In dimensional modeling, three primary schema patterns address different analytic requirements and trade-offs:

A **star schema** centers on a single fact table—storing quantitative measures—surrounded by denormalized dimension tables that hold all descriptive attributes needed for slicing and dicing. By eliminating the need for deep joins, star schemas deliver very fast query performance, making them ideal for interactive dashboards and ad-hoc reporting. The trade-off is data redundancy: the same attribute values may be repeated across multiple dimension rows.

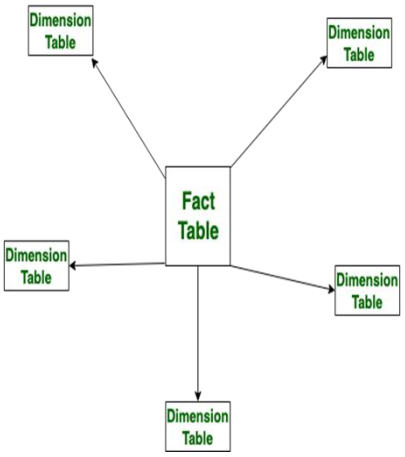

Figure 2.2: Star Schema Diagram

A **snowflake schema** refines the star approach by normalizing its dimensions into related sub-tables. For instance, a geographical dimension might split into separate Country, State, and City tables linked by foreign keys. This reduces storage requirements and enforces hierarchical consistency, but each additional join adds latency to query execution—particularly in deep hierarchies. Snowflake schemas are thus more storage-efficient than stars but incur slower query performance.

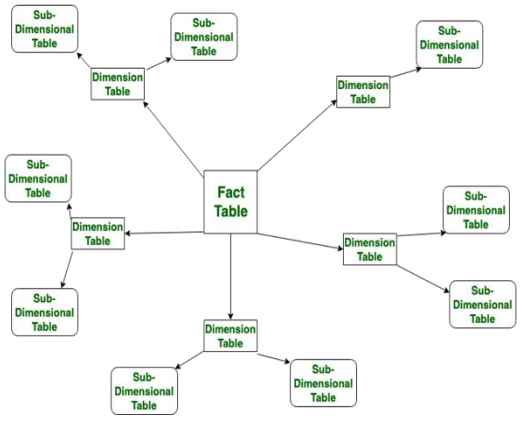

Figure 2.3: Snowflake Schema Diagram

At the enterprise level, a **galaxy schema** (or fact constellation) supports multiple fact tables that share conformed dimensions. This design lets diverse business processes—such as sales, inventory management, and finance—reuse the same dimensions (e.g., DateDim, CustomerDim) across different analyses. While galaxy schemas maximize consistency and avoid duplicate dimension definitions, they introduce greater design and maintenance complexity, since updates to a shared dimension can ripple through many fact tables.

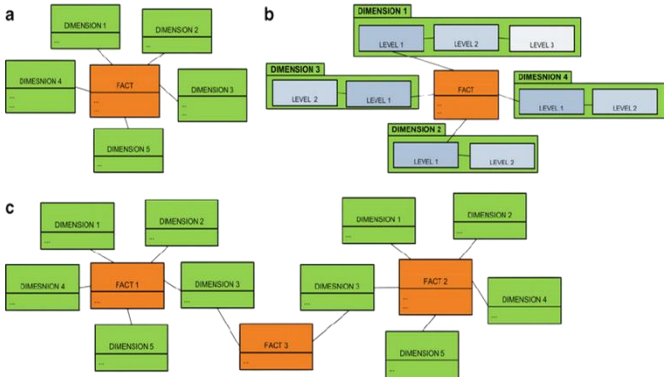

Figure 2.4: Galaxy Schema Diagram

Table 2-1: Schema Design Patterns

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

Comparison: 3NF vs. Dimensional Modeling

Table 2-2: 3NF vs. Dimensional Modeling

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

DataForge prioritizes dimensional schemas to ensure rapid, accurate analytics.

2.2.3 Fact and Dimension Tables

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

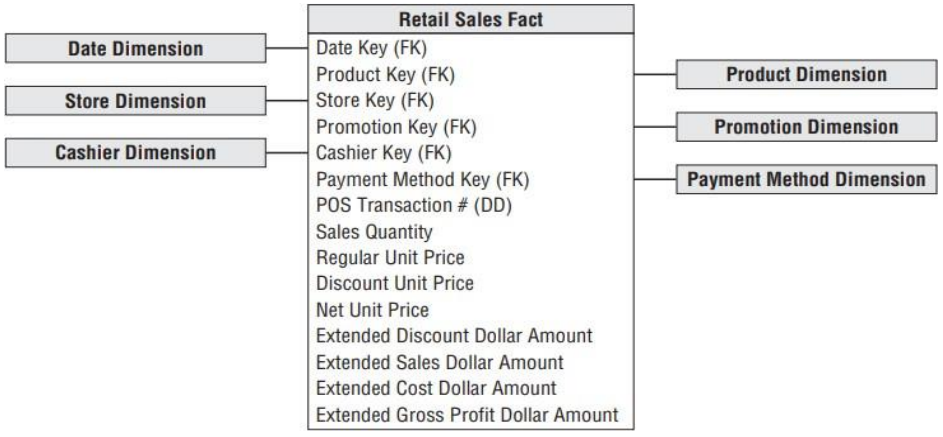

Figure 2.5: Fact Table Structure (Source: The Data Warehouse Toolkit - Kimball).

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

Table 2-3: Sample Date Dimension

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

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

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

Original row in Product dimension:

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

Updated row in Product dimension:

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

Figure 2.6: Overwrite outdated Type 1 (Source: The Data Warehouse Toolkit - Kimball).

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 2.7: Slowly Changing Dimension Type 2 (Source: The Data Warehouse Toolkit - Kimball).

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 2.8: Add a new column for old values Type 3 (Source: The Data Warehouse Toolkit - Kimball).

Figure shows how SCD Type 3 (e.g., retain previous category).

Table 2-4: Procedure And Usecases

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

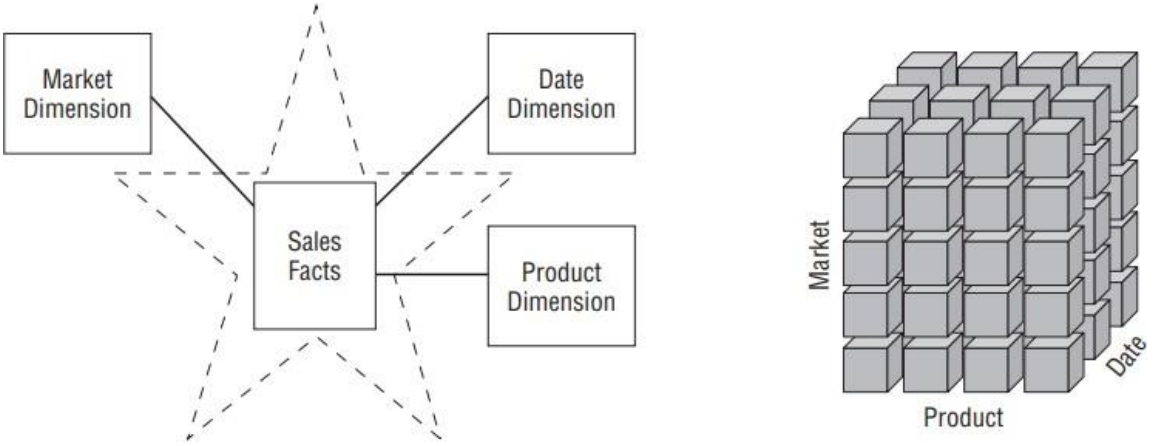

Figure 2.9: Retail Sales Star Schema (Source: The Data Warehouse Toolkit - Kimball). Depicts a central Sales_Fact table linked to Product_Dim, Customer_Dim, Store_Dim, and Date_Dim.

2.2.4 Grain & Additivity Rules

- **Grain Definition:** Precisely specify the level of detail—e.g., one row per order line versus one row per order header.
- **Additivity Classification:**
  - **Additive:** Sum across all dimensions (e.g., sales amount).
  - **Semi-Additive:** Summable across some dimensions only (e.g., account balance).
  - **Non-Additive:** Cannot meaningfully aggregate (e.g., ratios).

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

2.2.5 Conformed Dimensions & Naming Conventions

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

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

2.2.6 Data Warehouse Bus Architecture

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

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

Figure 2.10: Data Warehouse Bus Matrix

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.

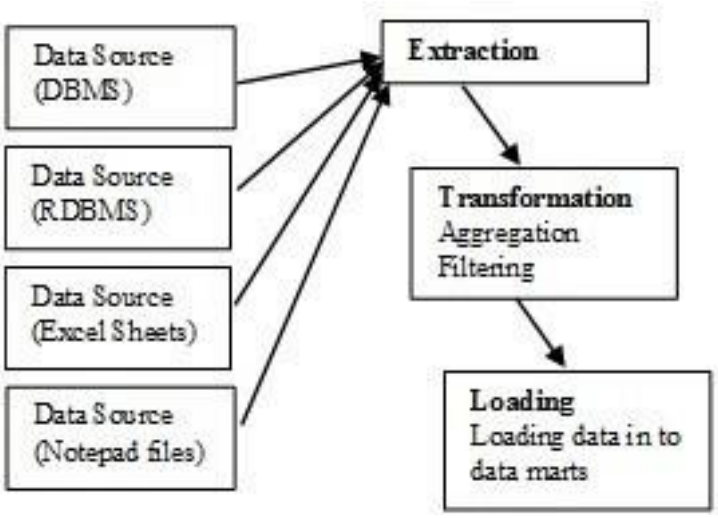

Figure 2.11: ETL Process Flow
Shows the three stages—Extract, Transform, Load—from source systems into the data warehouse.

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 2.12: 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

Efficient query performance and cost-effective storage are critical for large-scale data warehouses. DataForge's analytics module analyzes table statistics—row counts, data volume, column cardinality, and historical query logs—to automatically recommend the following optimizations:

- **Indexes**

  DataForge examines past query patterns, identifying the most frequently filtered or joined columns. It then suggests appropriate index types: **clustered indexes** on tables where range scans dominate (e.g., date-based filters on fact tables), and **non-clustered indexes** for high-cardinality lookup columns in dimension tables. Recommendations include the exact index

DDL (columns, sort order) and an estimated performance gain based on simulated query plans.

- **Partitioning**

  For very large fact tables, DataForge proposes **range partitioning** (commonly by date) or **list partitioning** (by region, product line, or other categorical attributes) to prune data scans. It analyzes historical query predicates to determine optimal partition keys and boundary definitions, then generates the CREATE PARTITION scripts. By restricting each query to a subset of partitions, disk I/O is dramatically reduced, often cutting scan times by 50–90%.

- **Materialized Aggregates**

  High-frequency roll-up queries—such as daily or monthly summaries—can be accelerated by **precomputing and storing aggregate tables**. DataForge identifies "hot" aggregation patterns from query logs (e.g., total sales by store per day) and suggests corresponding MATERIALIZED VIEW definitions. It also schedules automated refresh jobs (incremental where possible), balancing freshness requirements against compute cost.

- **Compression**

  Storage footprint and I/O throughput can be improved through **row-level** or **page-level** compression. DataForge profiles column data distributions—identifying low-cardinality text fields for dictionary encoding and high-repetition numeric columns for run-length encoding—and recommends the most effective compression scheme for each table or partition. It provides the necessary ALTER TABLE ... SET COMPRESSION commands along with projected space savings and potential CPU overhead.

By integrating these recommendations directly into the schema editing workflow, DataForge enables data engineers to apply performance and storage optimizations at design time, ensuring warehouses not only model data correctly but also serve queries efficiently and economically.

2.2.11 AI in Data Warehousing

Artificial intelligence augments every stage of schema generation and optimization in DataForge, providing domain-aware insights and automated quality checks:

- **Domain Detection**

  DataForge uses a hybrid approach combining **TF-IDF vectorization** of table and column names with **pre-trained embedding similarity** (e.g., BERT-based models). By comparing

the input schema's terminology and metadata against domain-specific corpora (retail, finance, healthcare, etc.), it assigns a domain label with confidence scores. This enables downstream suggestions—such as industry-standard fact and dimension patterns—to be tailored precisely to the user's business context.

- **Schema Enhancement**

  Once the domain is established, DataForge invokes an LLM to suggest missing elements: common dimension tables (e.g., `Promotion_Dim` for retail), essential audit fields (`created_at`, `updated_at` on all fact tables), and compliance-related columns (e.g., GDPR consent flags). The AI generates structured proposals—complete with column definitions, data types, and constraint clauses—that DataForge parses into actionable JSON suggestions for user review.

- **Anomaly Detection**

  Leveraging clustering on column metadata (data type, value distribution, null ratios) and semantic embedding outlier detection, DataForge flags suspicious definitions. Examples include a numeric "date" column stored as text, inconsistent naming patterns within the same dimension, or an unusually large number of foreign keys on a candidate dimension table. Each anomaly is highlighted in the UI with an explanatory message and, where possible, an automated fix recommendation.

- **Performance Recommendations**

  Beyond basic optimizations, DataForge's AI analyzes query logs and execution plans through a feedback loop: it identifies slow-running queries or frequently scanned columns and recommends targeted improvements. These can include composite indexes on multi-column predicates, selective denormalization of dimension attributes into the fact table, or dynamic partition pruning strategies. By continuously learning from operational telemetry, the system evolves its suggestions to match real-world usage patterns.

Together, these AI-driven capabilities ensure that DataForge not only automates the mechanical aspects of schema creation but also brings deep, context-aware expertise to bear—bridging the gap between raw DDL and production-ready, high-performance data-warehouse architectures.

2.3 Previous Studies and Works

This section surveys foundational approaches to automated schema generation, grouping prior research into three categories: heuristic and cost-based algorithms, graph-neural network models, and rule-based domain heuristics. For each work, we provide a concise description of the methodology and, where available, outline the high-level system architecture.

2.3.1 Academic Research on Automated Schema Generation

1. **Heuristic and Cost-Based Algorithms**

Abadi et al. (2016) propose a three-phase, unsupervised pipeline for generating fully normalized relational schemas from semi-structured input.

1. **Dependency Mining:** They start by scanning attribute co-occurrence patterns to identify "soft" functional dependencies—attributes that frequently appear together.
2. **Candidate Table Formation:** Next, they group strongly related attributes into provisional tables via a clustering step, using dependency scores as edge weights in a graph that they partition.
3. **Entity Merging:** Finally, overlapping clusters are merged if they share a high proportion of attributes, ensuring minimal redundancy in the final schema.

Their reference architecture consists of a **Dependency Extractor** module (processing raw JSON or CSV), a **Clustering Engine** (applying graph-partition algorithms), and an **Merger** component (resolving overlaps). While highly effective at producing 3NF designs, the approach assumes well-structured, complete input data and does not incorporate any business-domain knowledge or allow user feedback during generation.

FACT-DM Framework (EDBT 2024) adopts a cost-based model to generate denormalized (dimensional) schemas by optimizing multiple objectives—time to load, storage footprint, and query latency.

1. **Cost Model Definition:** They formalize costs for storage (based on row counts and compression ratios), load time (I/O throughput), and query performance (estimated scan costs).
2. **Search Algorithm:** A heuristic search explores possible fact/dimension layouts—merging or splitting tables—and evaluates each candidate against the cost model.
3. **Selection and Refinement:** The best-scoring schema is then fine-tuned via local moves (e.g., adding or removing a column from a dimension) to further balance the cost metrics.

Architecturally, FACT-DM comprises a **Cost Estimator** (calculating metrics from sample data), a **Schema Generator** (enumerating candidate designs), and an **Optimizer** (searching the design space). While it can scale to large datasets and offers tunable trade-offs, FACT-

DM lacks interactive correction mechanisms and does not interpret semantic meaning—limits that restrict its flexibility to static optimization.

2. **Graph-Neural Network Models**

Fey et al. (2023) introduce a GNN-based system that represents a database schema as a graph where nodes are tables and edges are primary-to-foreign-key relationships.

- **Graph Construction:** They convert each table's attribute vector (e.g., column data types encoded as features) into node embeddings, and connect nodes by schema relationships.
- **Message Passing:** A multi-layer GNN propagates information across the graph, allowing each table embedding to aggregate structural context from its neighbors.
- **Clustering & Classification:** The final node embeddings are clustered to reveal candidate dimension groups or passed through an MLP to predict whether a table should be a fact or a dimension.

Their system architecture consists of a **Feature Extractor** (building initial node features), a **GNN Engine** (performing message passing), and a **Post-Processing Module** (clustering and classification). This approach excels at capturing complex inter-table dependencies without manual feature engineering, but it requires substantial labeled schemas for training and significant compute resources at inference time.

Relational Graph Transformer (RelGT) — Dwivedi et al. (2025) extends GNNs with attention mechanisms to better capture long-range dependencies in large schemas.

- **Attention Layers:** By replacing traditional message passing with transformer-style self-attention, RelGT allows each table to attend to all others, learning weighted interactions beyond immediate neighbors.
- **Position Encoding:** They incorporate positional encodings derived from schema hierarchy (e.g., depth in a snowflake) to preserve structural information in the transformer.
- **Domain-Specific Fine-Tuning:** The model is pre-trained on a corpus of known schemas labeled by domain, then fine-tuned to infer fact/dimension assignments in new examples.

RelGT's architecture includes an **Embedding Module**, multiple **Transformer Blocks**, and a **Domain Classifier** head. While it generalizes across varying schema sizes, the approach demands large, diverse training datasets and high GPU memory—barriers for smaller teams or rapidly changing schema collections.

3. **Rule-Based Domain Heuristics**

   Elamin et al. (2017) developed a dedicated reverse-engineering engine to convert transactional databases into star schemas using rule sets grounded in dimensional modeling principles.

   - **Rule Set Definition:** They codify heuristics such as "tables with more than 70% numeric columns and at least two foreign keys are candidate fact tables" or "dimension tables must contain a user-readable label column."
   - **Processing Pipeline:** The engine applies rules in sequence—first identifying facts, then scanning remaining tables for dimension roles, and finally verifying conformed dimensions against a predefined list (e.g., Date, Geography).

   The system is structured into a **Rule Evaluator** (applying each heuristic), an **Schema Assembler** (linking fact and dimension tables), and a **Validation Module** (ensuring referential integrity). This interpretable approach runs in milliseconds on large schemas but struggles when source schemas don't match the assumed patterns or when domains evolve beyond the original rule set.

   Fong et al. (2010) combine attribute clustering with rule-based filters to generate OLAP schemas in star, snowflake, or galaxy topologies.

   - **Attribute Clustering:** They use k-means on attribute metadata (e.g., data type, distinct-value count) to form initial groups.
   - **Heuristic Refinement:** Predefined rules then split or merge clusters based on foreign-key relationships and cardinality thresholds, choosing the appropriate schema pattern.
   - **Pattern Selector:** A final decision logic selects star vs. snowflake vs. galaxy based on the density of inter-cluster links and desired query performance profile.

   Their architecture features a **Clustering Engine**, a **Heuristic Refinement Module**, and a **Pattern Selector** component. While this hybrid method achieves good precision with limited data, it can misclassify tables in unfamiliar domains and offers no built-in mechanism for user feedback or iterative refinement.

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

Accurate extraction of schema metadata from raw SQL DDL is a foundational requirement for automated warehouse design. Over the years, three main approaches have emerged—parser generators, AST libraries, and regex-based tools—each with distinct advantages and limitations.

1. Parser Generators (ANTLR, PLY)

Parser generators such as ANTLR (ANother Tool for Language Recognition) and PLY (Python Lex-Yacc) provide formal grammars for SQL that can handle the full complexity of vendor-specific dialects.

- **Architecture & Workflow**
    1. **Grammar Definition**: A complete SQL grammar—often running into thousands of lines—is written in the tool's grammar language, specifying tokens (keywords, identifiers, operators) and production rules for statements (CREATE TABLE, ALTER TABLE, etc.).
    2. **Lexer and Parser Generation**: The tool auto-generates a lexer (tokenizer) and parser code in the target language (Java, Python).
    3. **Parse Tree Construction**: When presented with a DDL script, the parser builds a concrete parse tree (or CST) that precisely represents every language construct, including nested sub-clauses.
    4. **Tree Walking & Extraction**: Custom listener or visitor classes traverse the parse tree to extract table names, column definitions, data types, constraint clauses, and vendor-specific options (e.g., ENGINE=InnoDB, CLUSTERED INDEX).
- **Strengths**
    - **Completeness**: Covers virtually all SQL syntax—including proprietary extensions from Oracle, SQL Server, or MySQL—ensuring that even complex DDL constructs are correctly understood.
    - **Accuracy**: By relying on a formal grammar, parser generators avoid the brittleness of pattern matching and correctly handle nested queries, quoted identifiers, and unusual formatting.
- **Limitations**
    - **Maintenance Overhead**: Keeping the grammar up to date with evolving SQL standards and vendor extensions demands significant ongoing effort, especially in large teams or open-source projects.
    - **Performance**: Full-tree parsing can be slower and more memory-intensive, making it less suitable for rapid, on-the-fly ingestion of very large DDL files.

2. AST Libraries (SQLAlchemy, jOOQ)

Abstract Syntax Tree (AST) libraries like SQLAlchemy (Python) and jOOQ (Java) offer built-in parsers that convert SQL strings into a programmatic AST, easing the extraction of schema elements.

- **Architecture & Workflow**
    1. Tokenization & Normalization: The library first tokenizes the input SQL and applies normalization passes—removing extraneous whitespace, standardizing keyword case, and stripping vendor-specific annotations.
    2. AST Generation: It then builds a structured AST where nodes represent syntactic constructs such as Table, Column, PrimaryKey, and ForeignKey.
    3. Programmatic Access: Developers invoke AST APIs (e.g., ddl = parser.parse(sql_string); for table in ddl.tables: …) to inspect table metadata, iterate over column definitions, and read constraint specifications.
- **Strengths**
    - Ease of Use: High-level APIs shield developers from low-level parsing details, enabling rapid extraction of schema components without manual grammar writing.
    - Integration: AST libraries are often part of broader ORM or query-building frameworks, allowing seamless round-tripping between code and database schemas.
- **Limitations**
    - Dialect Coverage: While mainstream SQL features are well-supported, proprietary extensions, edge-case DDL syntax, and custom functions may be ignored or result in parse errors.
    - Dependency Footprint: Including a full ORM or query DSL may introduce substantial dependencies into a lightweight schema-generation tool.

3. Regex-Based Tools (DDLParser, Custom Scripts)

Regular-expression–based parsers capture common DDL patterns by matching text sequences. Tools like DDLParser (Python) or in-house scripts exemplify this approach.

- **Architecture & Workflow**
    1. Pattern Library: A curated set of regex patterns is defined to match CREATE TABLE headers, column-definition lines (<column_name> <data_type> [constraints]), and simple FOREIGN KEY clauses.
    2. Line-by-Line Scanning: The parser scans the SQL file sequentially, applying each regex to extract table names, accumulate column lists, and detect key constraints.
    3. Post-Processing: Captured strings are normalized—trimming quotes, standardizing case—and assembled into a JSON-like structure ({ table: "Orders", columns: […], primaryKeys: […], foreignKeys: […] }).

- **Strengths**
  - Speed: Regex matching is extremely fast and lightweight, making it suitable for interactive tools and large-scale batch processing.
  - Simplicity: Developers can extend or tweak the pattern library without delving into complex grammar files or AST APIs.
- **Limitations**
  - Brittleness: Unexpected formatting (line breaks, embedded comments), nested parentheses, or vendor-specific clauses can break regex patterns, leading to missed or malformed extractions.
  - Limited Coverage: Complex DDL features—composite keys, check constraints, partition definitions—often require custom patterns or are skipped entirely.

Relevance to DataForge

To balance performance, coverage, and maintainability, DataForge adopts a **hybrid parsing architecture**:

1. **Primary Regex Layer**
   - Rapidly ingests the majority of DDL constructs via a comprehensive regex library tuned to common enterprise patterns.
   - Handles simple `CREATE TABLE`, column lines, and basic key constraints in milliseconds.
2. **Selective AST Refinement**
   - For statements flagged as "unsupported" or those containing nested or vendor-specific syntax, DataForge invokes a lightweight AST parser (using `sqlparse` under the hood) to recover missing metadata.
   - This fallback ensures 95%+ extraction accuracy without imposing the performance penalty of full AST parsing on every statement.
3. **Normalization & Validation**
   - Parsed identifiers flow through a normalization pipeline—stripping quotes, enforcing naming conventions, and unifying data-type synonyms (e.g., `INT4` → `INTEGER`).
   - An integrated validation engine cross-checks extracted keys and constraints, highlighting inconsistencies for user review.

By combining the speed of regex-based ingestion with the robustness of AST parsing for edge cases, DataForge achieves both high performance and comprehensive coverage, laying a solid foundation for downstream schema classification and AI-driven enhancements.

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

A number of mature ETL and data-integration platforms dominate the enterprise landscape. While they excel at data movement, transformation, and metadata management, none provide the end-to-end, AI-driven schema-generation capabilities that DataForge delivers. Below we examine five leading solutions in detail.

Talend Data Integration

Talend's open-source and commercial offerings provide a code-free, graphical interface for building ETL jobs. At its core is a repository of prebuilt components ("connectors") that extract data from sources (databases, files, APIs), transform it via drag-and-drop routines (filter, join, aggregate), and load it into targets. The platform includes a visual schema mapper to reconcile source and target data models, but all mapping logic must be defined manually. Talend offers lineage tracking and impact analysis—capturing metadata for data governance—but lacks any AI-driven domain inference or automated suggestion of dimensional constructs. Users can reverse-engineer an existing database into an ER diagram, yet transforming that into a star or snowflake schema remains a hands-on task.

Informatica PowerCenter

Informatica PowerCenter is widely deployed for large-scale data integration, offering robust connectivity, high-availability clustering, and enterprise metadata management. Its core engine orchestrates parallel ETL workflows with fine-grained performance tuning, while the Metadata Manager provides a searchable catalog of schema artifacts, transformations, and data lineage. PowerCenter includes a Model Designer for drawing ER diagrams and defining table mappings, but all dimensional modeling (fact/dimension designation, slowly changing dimension rules) must be specified by the developer using template widgets. There is no built-in AI component to suggest schema optimizations or infer business contexts; instead, Informatica focuses on stability, scalability, and declarative change-data-capture capabilities.

Microsoft SQL Server Integration Services (SSIS)

SSIS ships with SQL Server and integrates tightly into Microsoft's BI stack. Packages are built in Visual Studio using a toolbox of data flow and control flow tasks. SSIS simplifies common patterns—bulk load, lookups, fuzzy matching—and can auto-generate simple mappings when source and destination columns match by name. Its Project-Deployment Model supports parameterization and environment-specific configurations. However, SSIS does not include any AI-

powered schema design intelligence: fact vs. dimension classification, audit-field recommendations, or domain inference are outside its remit. Schema generation relies on manual table definitions, and there is no mechanism to automatically evolve or optimize the dimensional model based on usage patterns.

ER/Studio

ER/Studio by IDERA is a dedicated data modeling IDE for enterprise architects. It excels at reverse-engineering existing databases into rich entity-relationship diagrams, supports forward engineering of logical and physical models, and enforces naming and normalization rules via built-in standards libraries. ER/Studio's Repository centralizes metadata, enabling team collaboration on schema changes. Despite its modeling power, ER/Studio remains a manual tool: defining fact tables, dimension tables, surrogate keys, and slowly changing dimension attributes all require user action. There is no automated AI component to infer the optimal dimensional design or suggest enhancements—users rely on the tool's rule-based validation to flag naming or referential inconsistencies.

DBT (Data Build Tool)

DBT has revolutionized analytics engineering by treating transformations as version-controlled SQL code. Users define staging, intermediate, and marts models in .sql files, and dbt handles dependency resolution, incremental builds, and documentation generation. **DBT** Cloud provides lineage graphs and catalog metadata, automatically documenting column descriptions, tests (uniqueness, not null), and data freshness. However, **DBT** assumes that the dimensional schema (tables and views) already exists: it does not reverse-engineer schemas nor suggest fact/dimension splits. All table creation logic must be written in SQL by the developer; **DBT**'s intelligence lies in testing and documentation, not in automated schema inference or AI-driven design.

Relevance to DataForge

While Talend, Informatica, SSIS, ER/Studio, and dbt each excel at specific facets of ETL, metadata management, and analytics engineering, they share a common limitation: schema design remains a manual, developer-led activity with no built-in AI guidance. None combine DDL parsing, heuristic classification, domain inference, and interactive dimensional modeling into a single, automated workflow. DataForge fills this gap by integrating each of those capabilities—extracting raw SQL, proposing fact/dimension roles, leveraging AI for domain-aware enhancements, and presenting a live, editable schema graph—thus streamlining the entire design process from raw DDL to production-ready warehouse schema.

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.

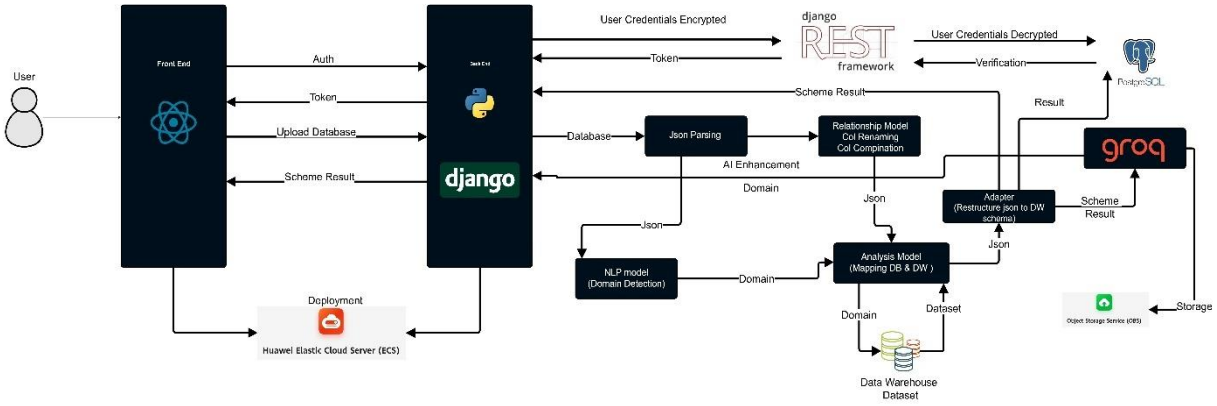

Figure 3.1: DataForge System Architecture

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:

Table 3-1: Functional Requirements

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

3.4 Nonfunctional Requirements

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

Table 3-2: Nonfunctional Requirements

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

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

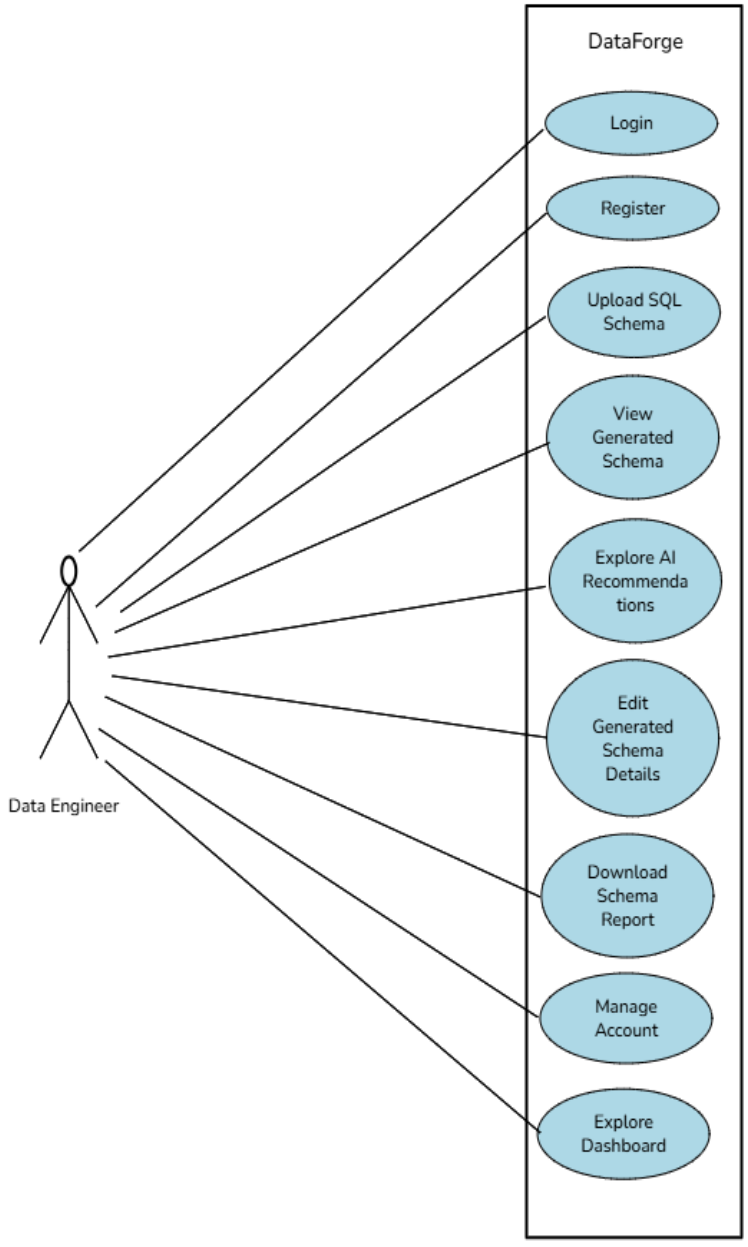

Figure 3.2: Use Case Diagram

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.

3.5.2 Class Diagram

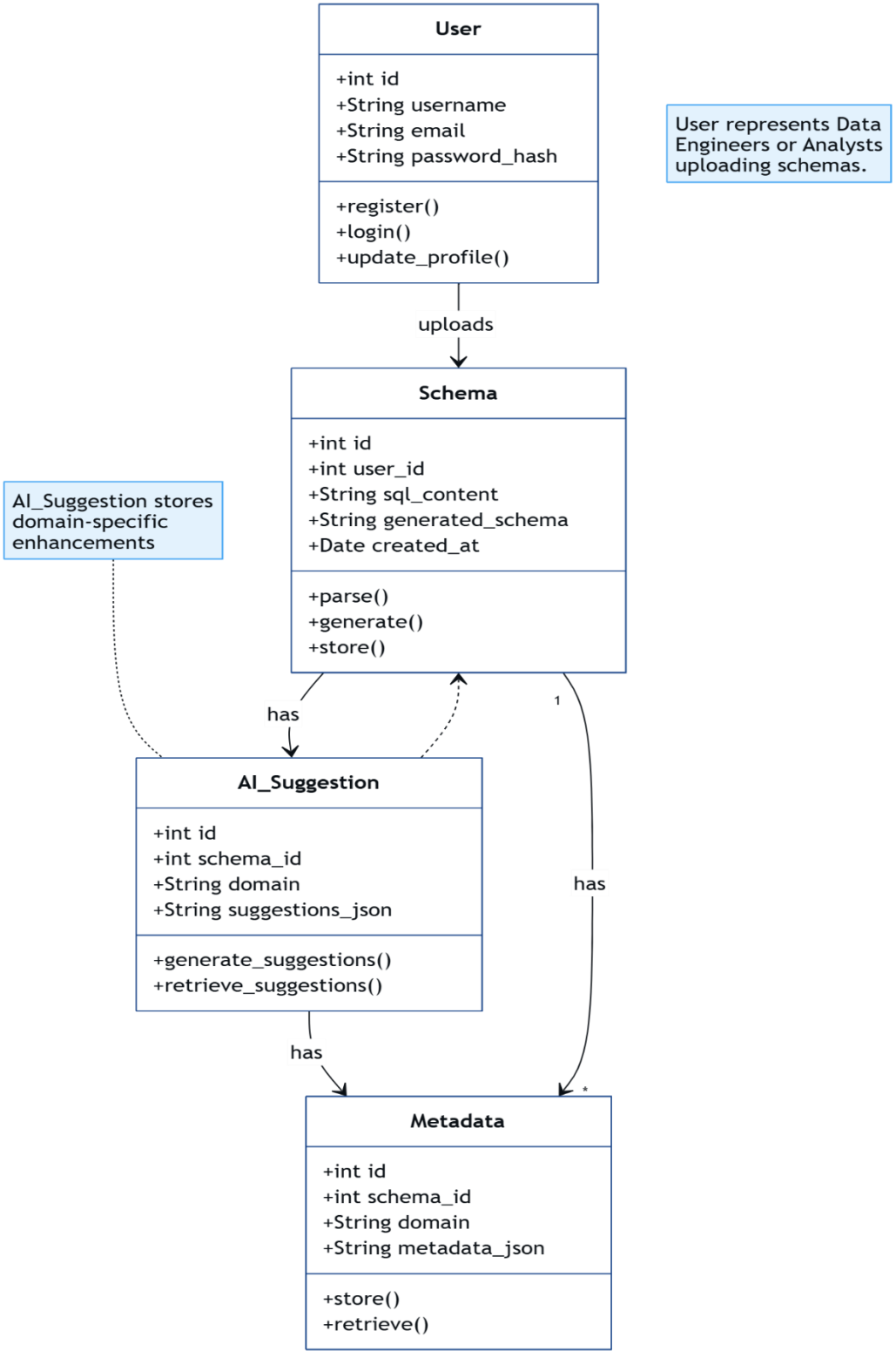

Figure 3.3: Class Diagram

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()
  - **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

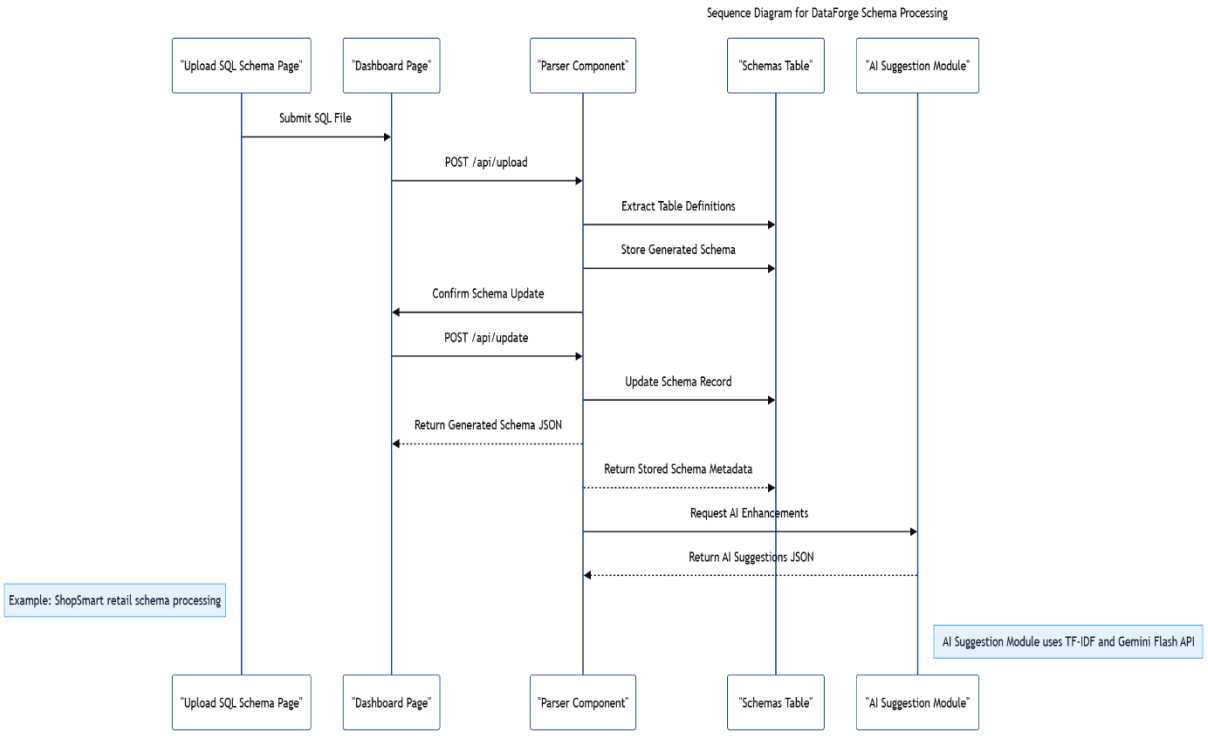

Figure 3.4: Sequence Diagram

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

3.5.4 Database Diagram

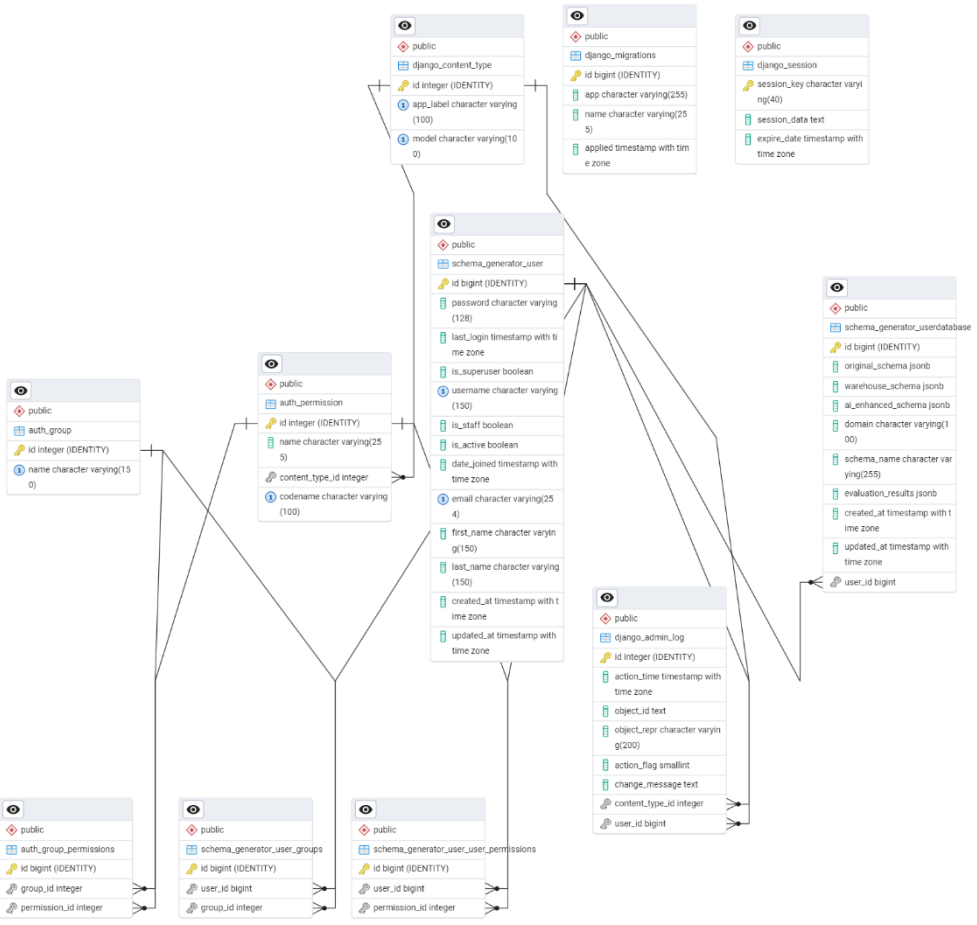

Figure 3.5: Database Diagram

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).

3.5.5 DataForge Workflow

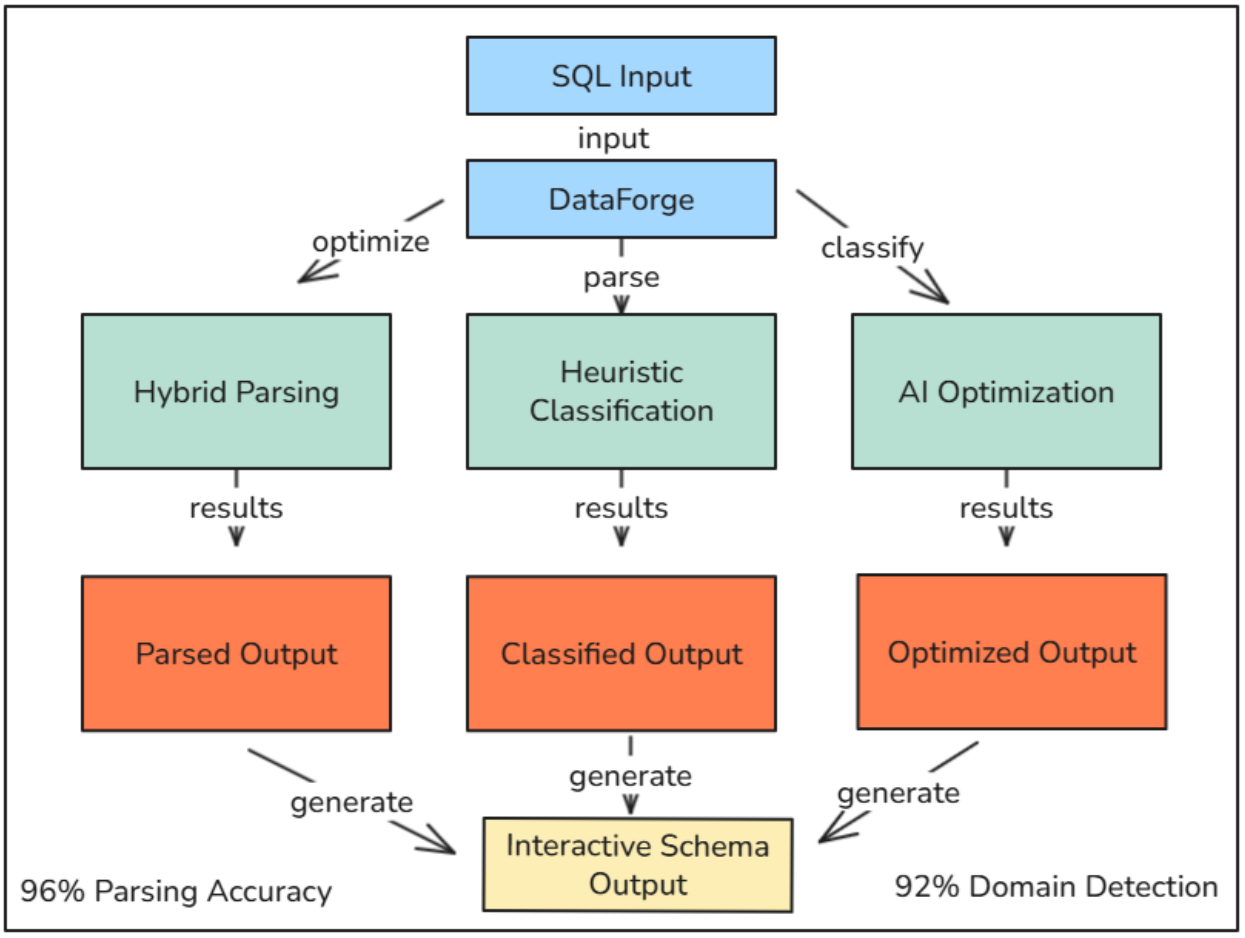

Figure 3.6: DataForge Workflow

The **DataForge** Workflow shows the system's data processing pipeline, beginning with an SQL Input entering the DataForge engine. The engine uses Hybrid Parsing to extract and structure data, Heuristic Classification to categorize tables, and AI Optimization to enhance schemas. These steps produce Parsed Output, Classified Output, and Optimized Output, which are combined to create the final Interactive Schema Output. The diagram emphasizes the iterative optimization and classification processes for generating a robust, domain-aligned data warehouse schema.

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.

3.6.1 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.

3.6.2 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.

3.6.3 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.

3.6.4 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.

3.6.5 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 v ıe datasets, software, and hardware used, then detail the core implementation and results.

4.1 Materials and Environment

4.1.1 Datasets

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

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

Orange Digital Center	Orange DC Internship Project	Sales	6	Educational DW project; includes customer, services, and performance logs.

4.1.2 Software and Frameworks

Table 4-2: Software and libraries employed.

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.**

4.1.3 Hardware & Cloud Configuration

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

Table 4-3: Hardware and Huawei Cloud instances.

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.

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): FK cols/total ≥ 0.5 → fact.
- **Numeric-Column Ratio:** ≥ 0.6 numeric fields → reinforce fact classification.
- **Cardinality Threshold:** Dimension keys ≤ 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** ≥ 95 %
2. **Domain Detection** ≥ 90 %
3. **Schema Generation Time** < 5 s (100 tables)
4. **Visualization Load** < 3 s (50 nodes)
5. **User Satisfaction** ≥ 4.0/5.0

4.3.2 End-to-End Metrics

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

Metric	Target	ShopSmart	TPC-DS	Prior Work
Parsing Accuracy (%)	≥ 95	96	95	~ 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	—

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 ± 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 % → 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 % → 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:

Table 4-5: Algorithmic evaluation results.

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 %

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

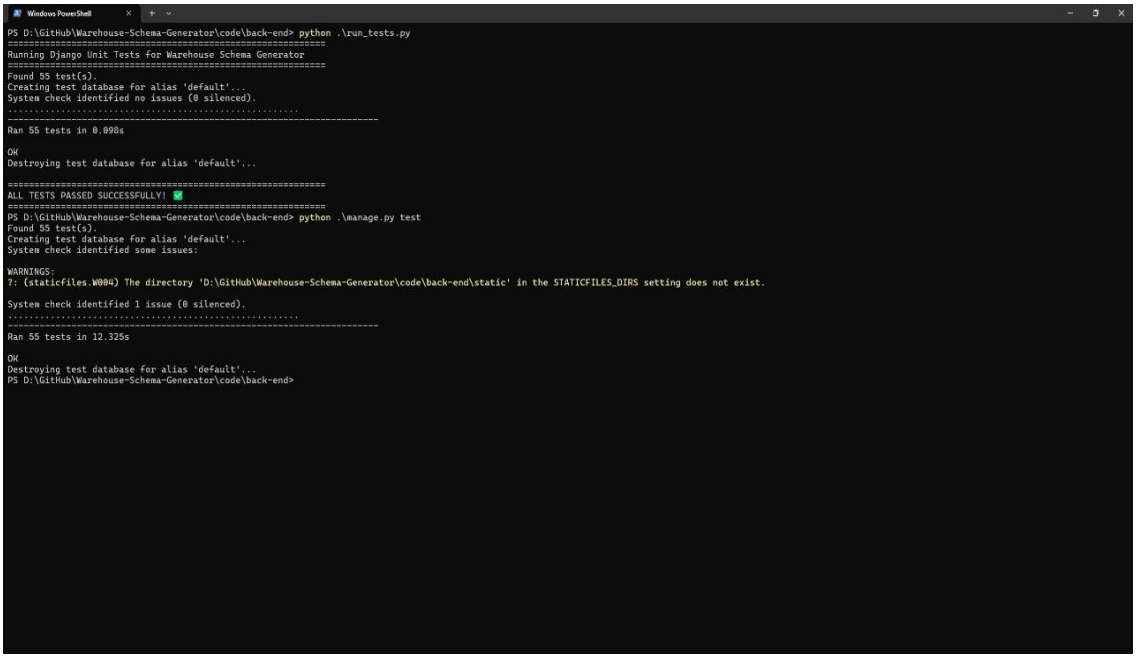

Figure 4-3: Unit Testing

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

Table 4-6: Challenges and resolutions.

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

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 login, features and Brief Information about DataForge

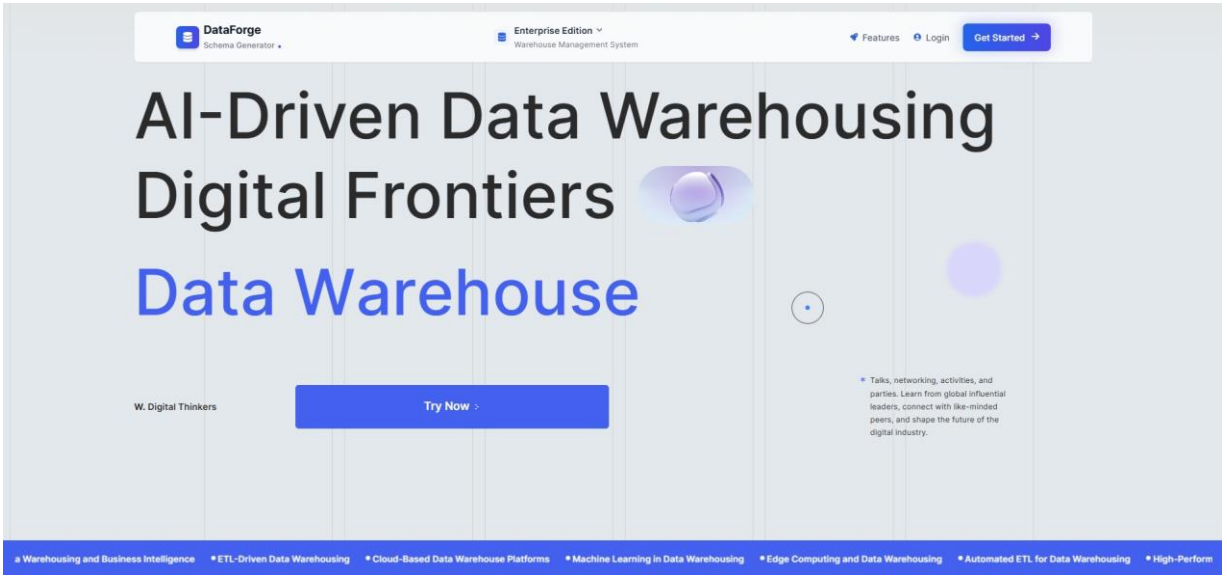

Figure 5.1: Landing Page

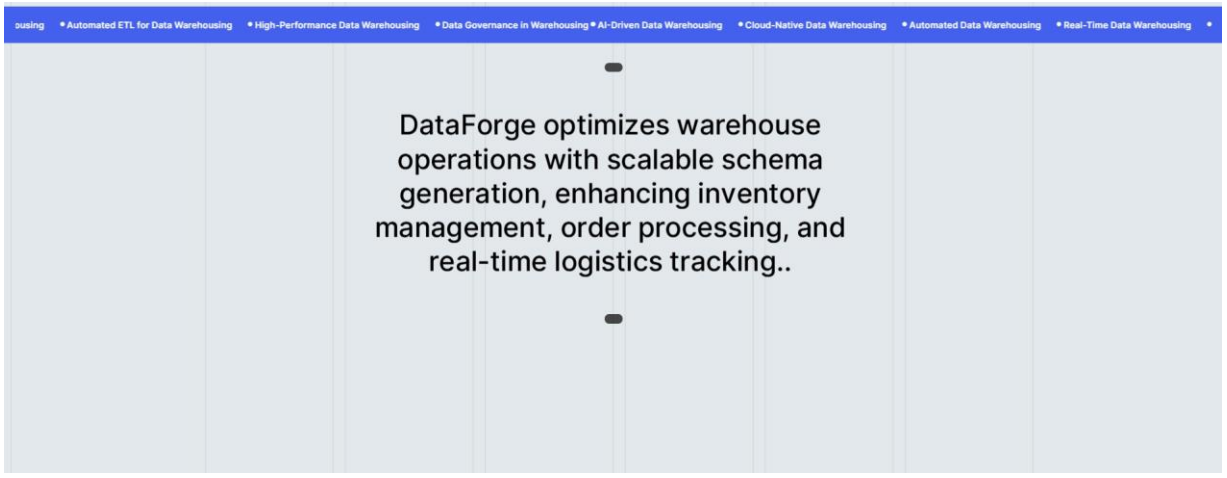

Figure 5.2: Landing Page II

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.

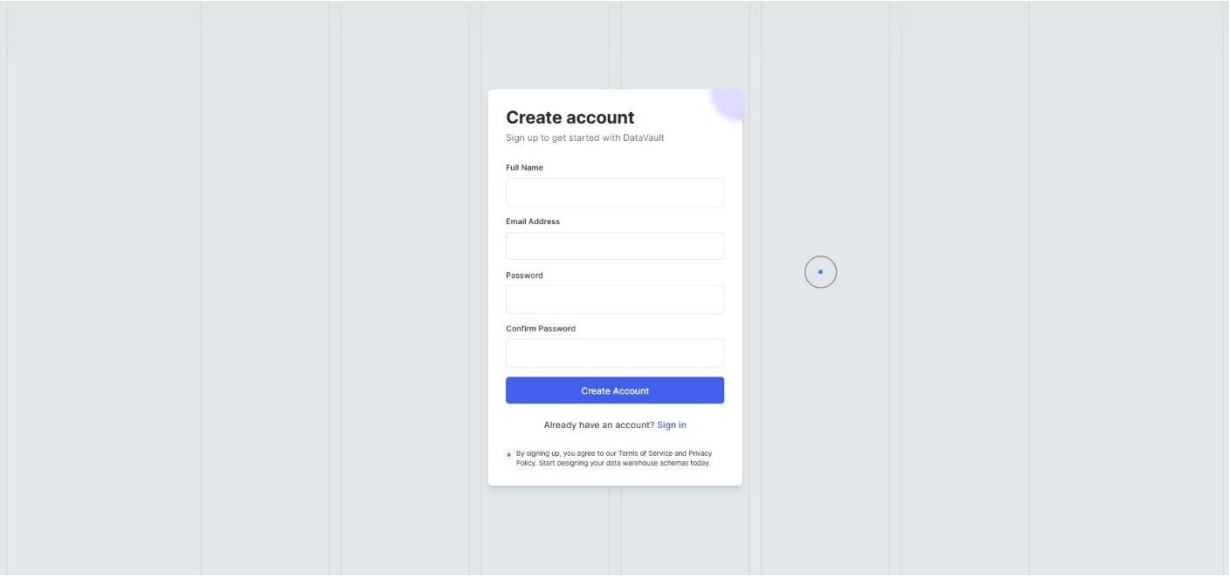

Figure 5.3: User Registration

5.3.3 Login Page

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

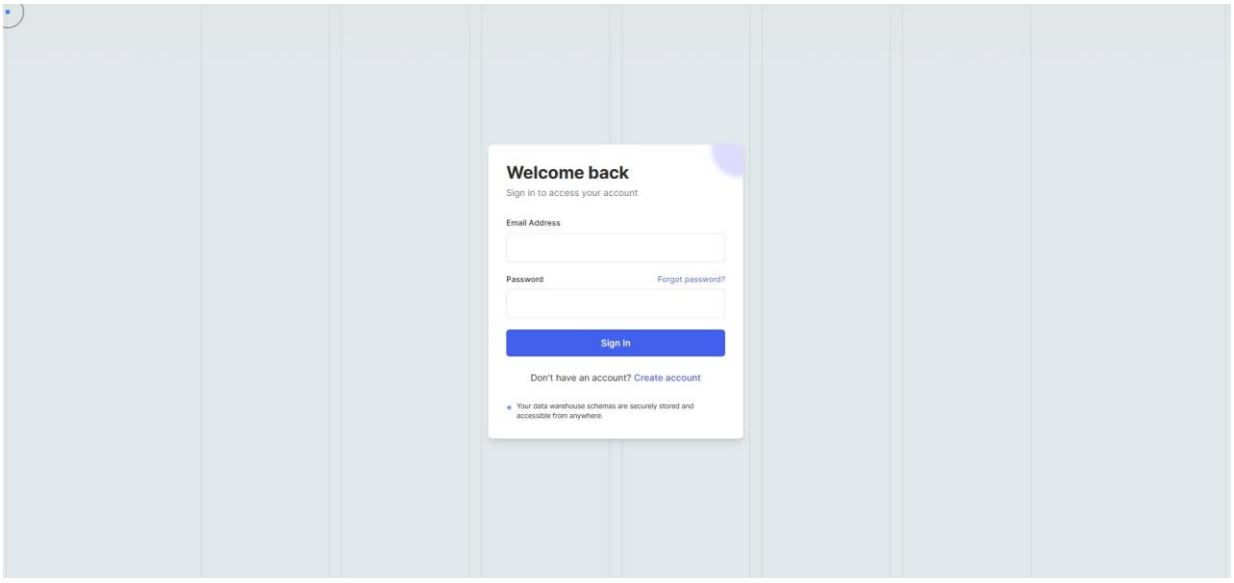

Figure 5.4: Login Page

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.

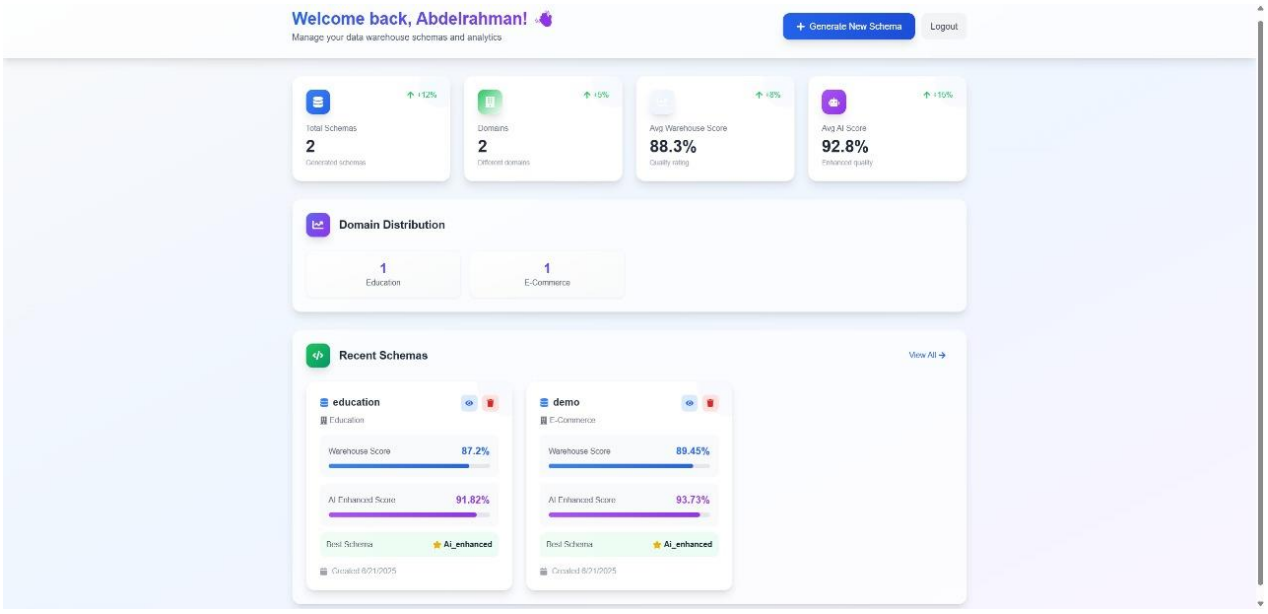

Figure 5.5: Dashboard

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.

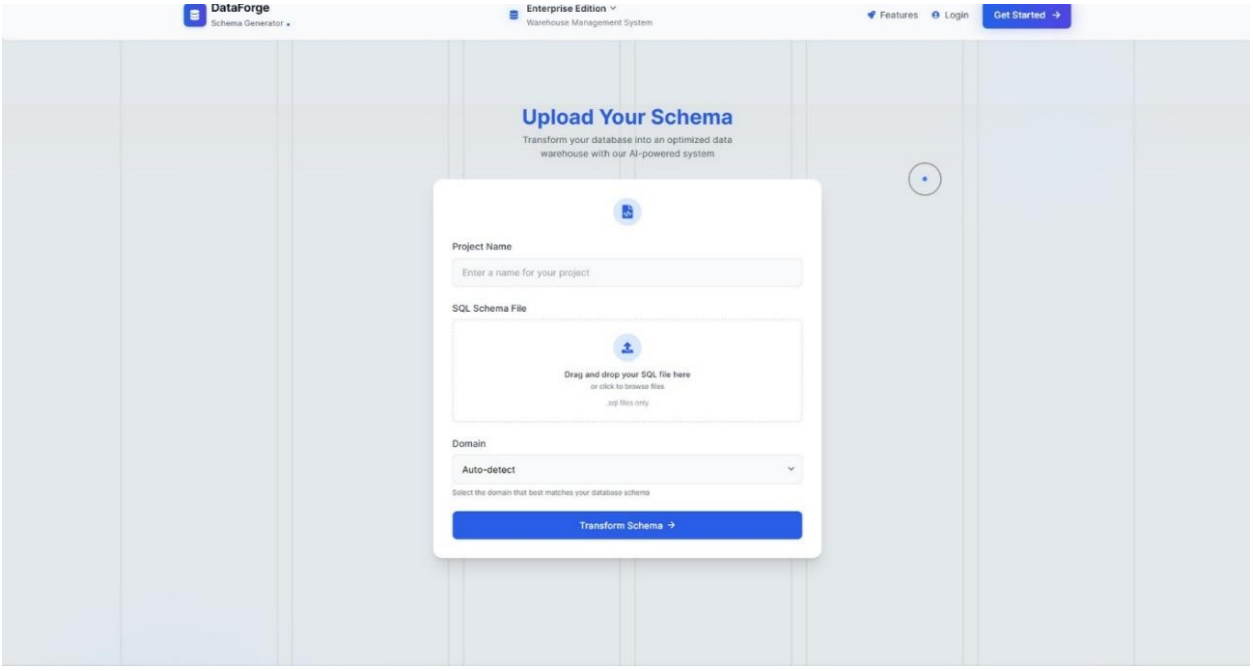

Figure 5.6: Upload SQL Schema

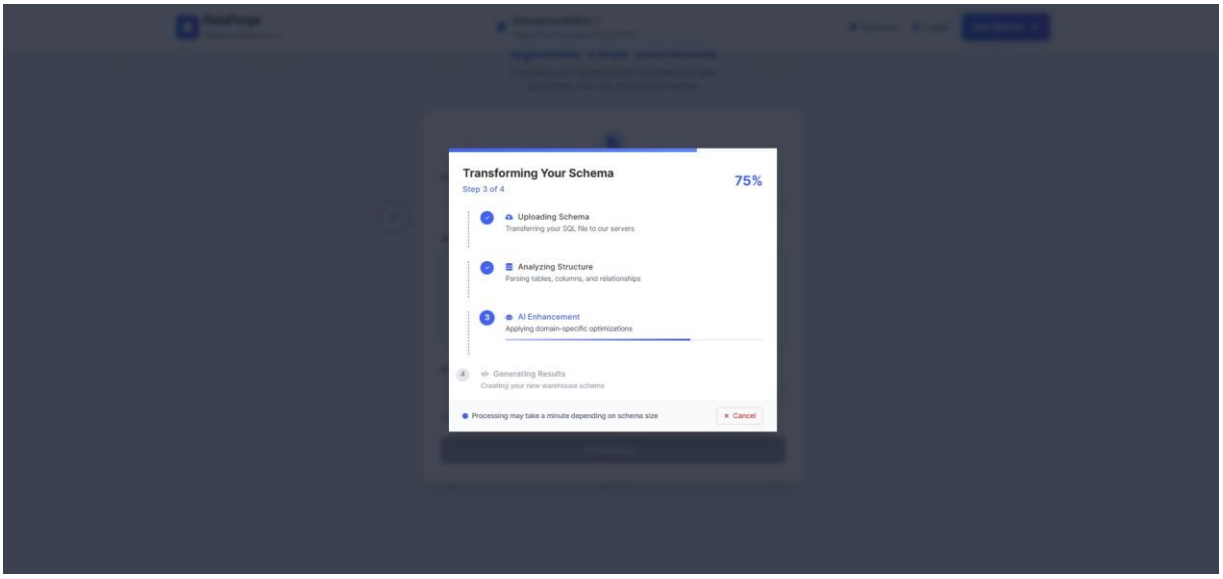

Figure 5.7: User Feedback

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.

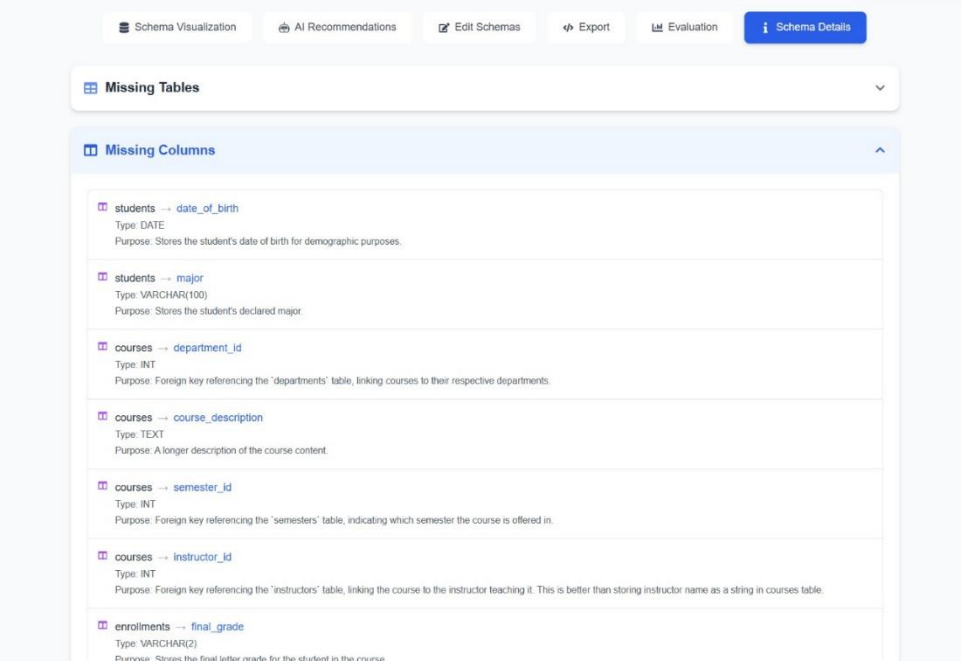

Figure 5.8: Upload Schema Details

5.3.7 View Uploaded Schema

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

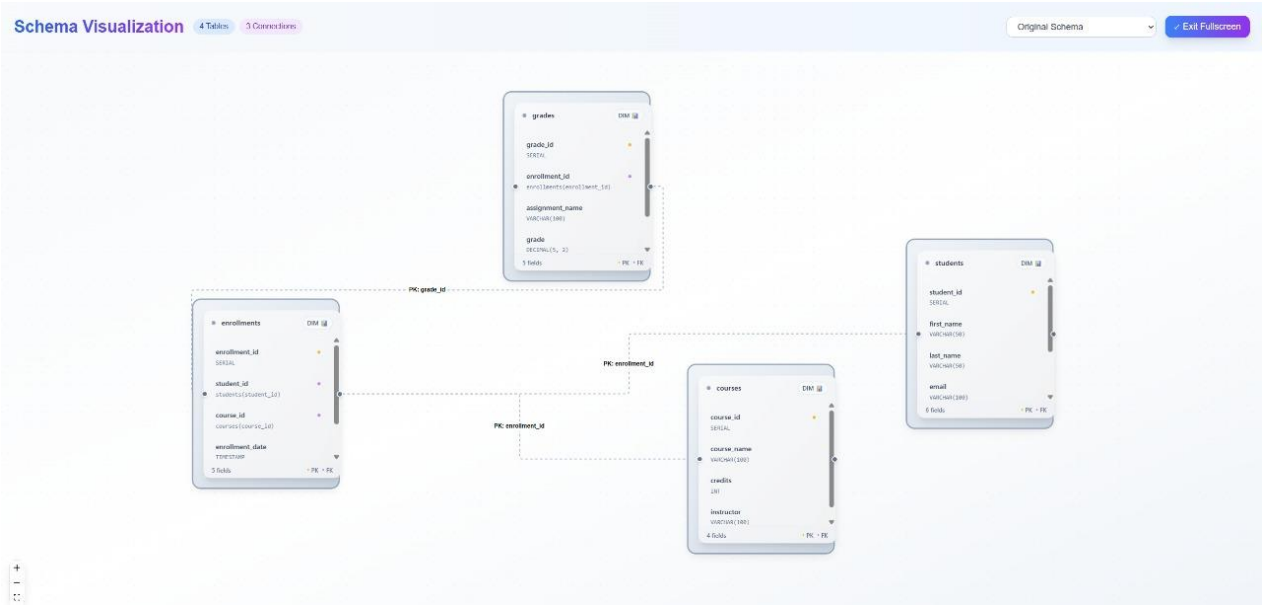

Figure 5.9: View Uploaded Schema

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.

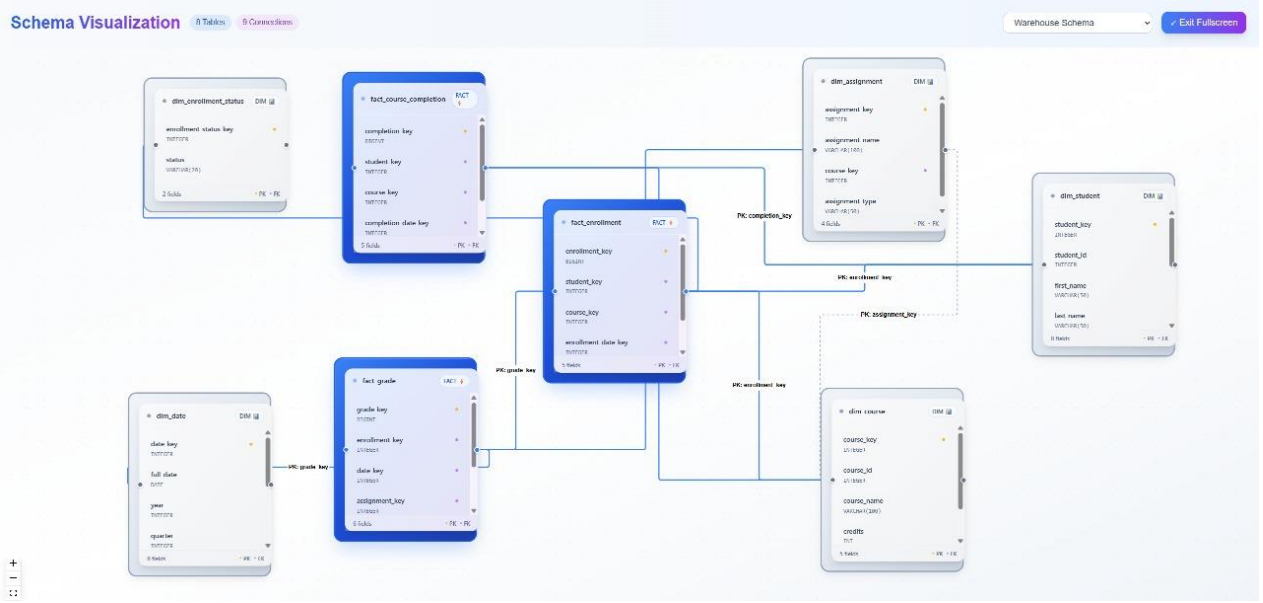

Figure 5.10: View Generated Schema Using Algorithms

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.

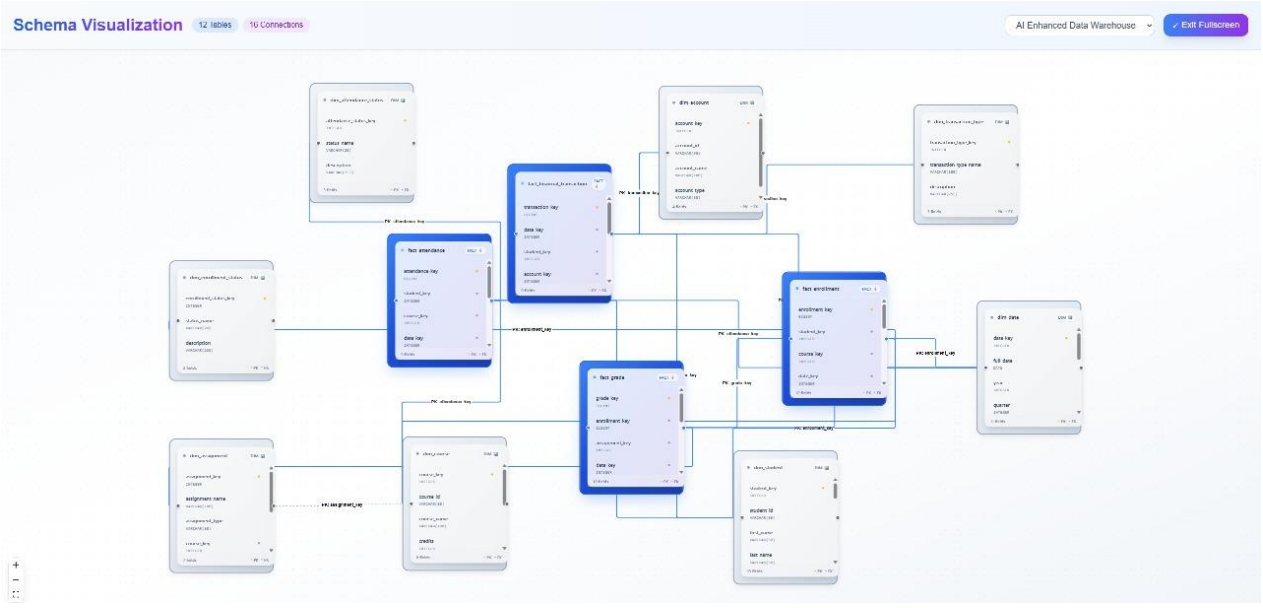

Figure 5.11: View Generated Schema Using AI Enhancement

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.

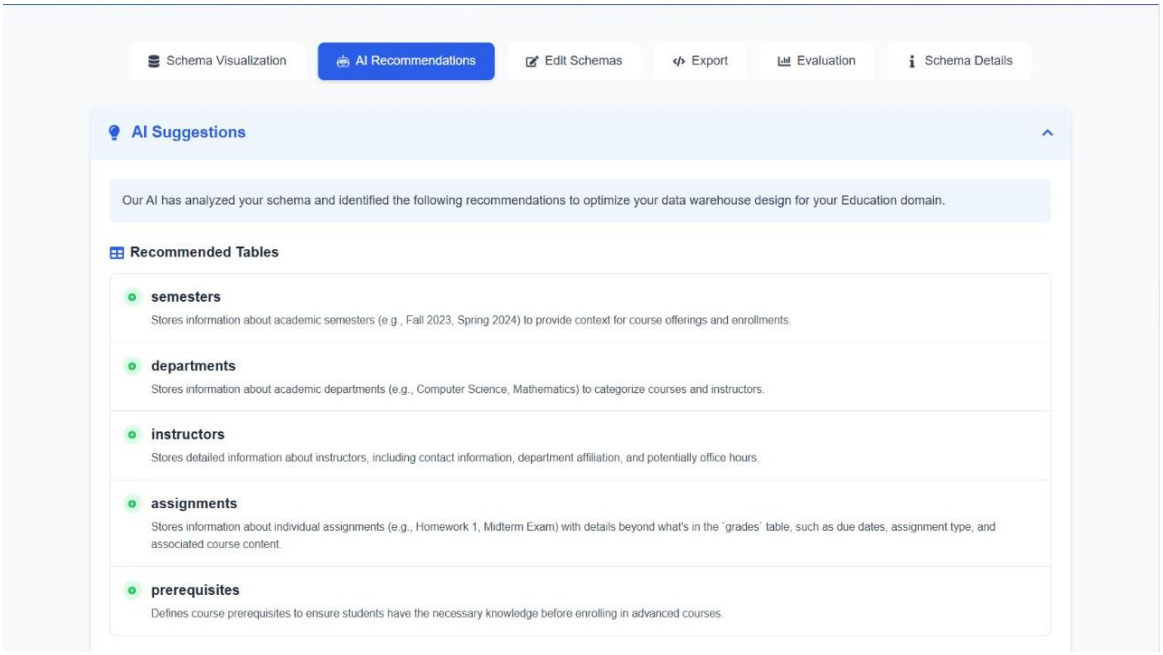

Figure 5.12: Explore AI Recommendations

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.

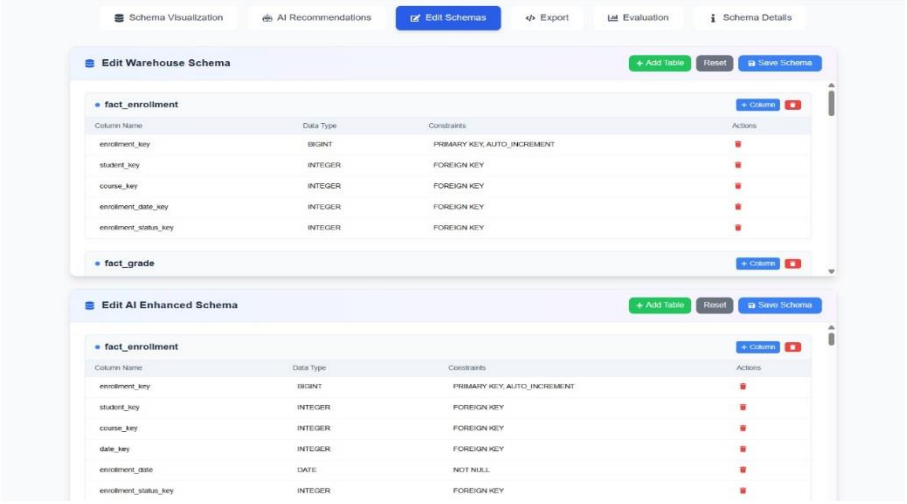

Figure 5.13: Edit Generated Schema

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.

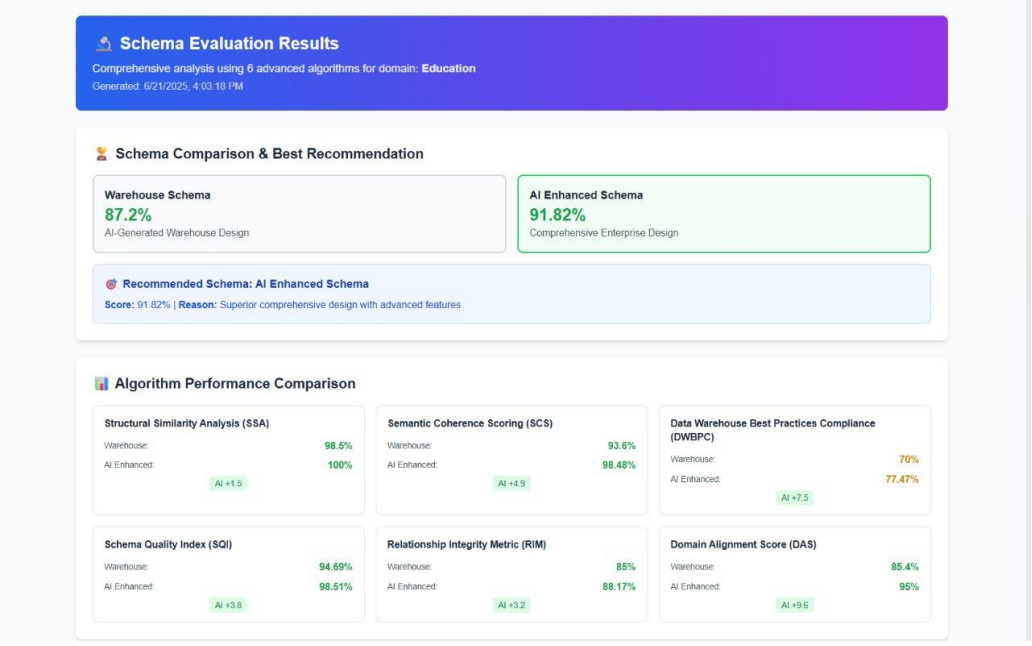

Figure 5.14: Schema Evaluation

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.

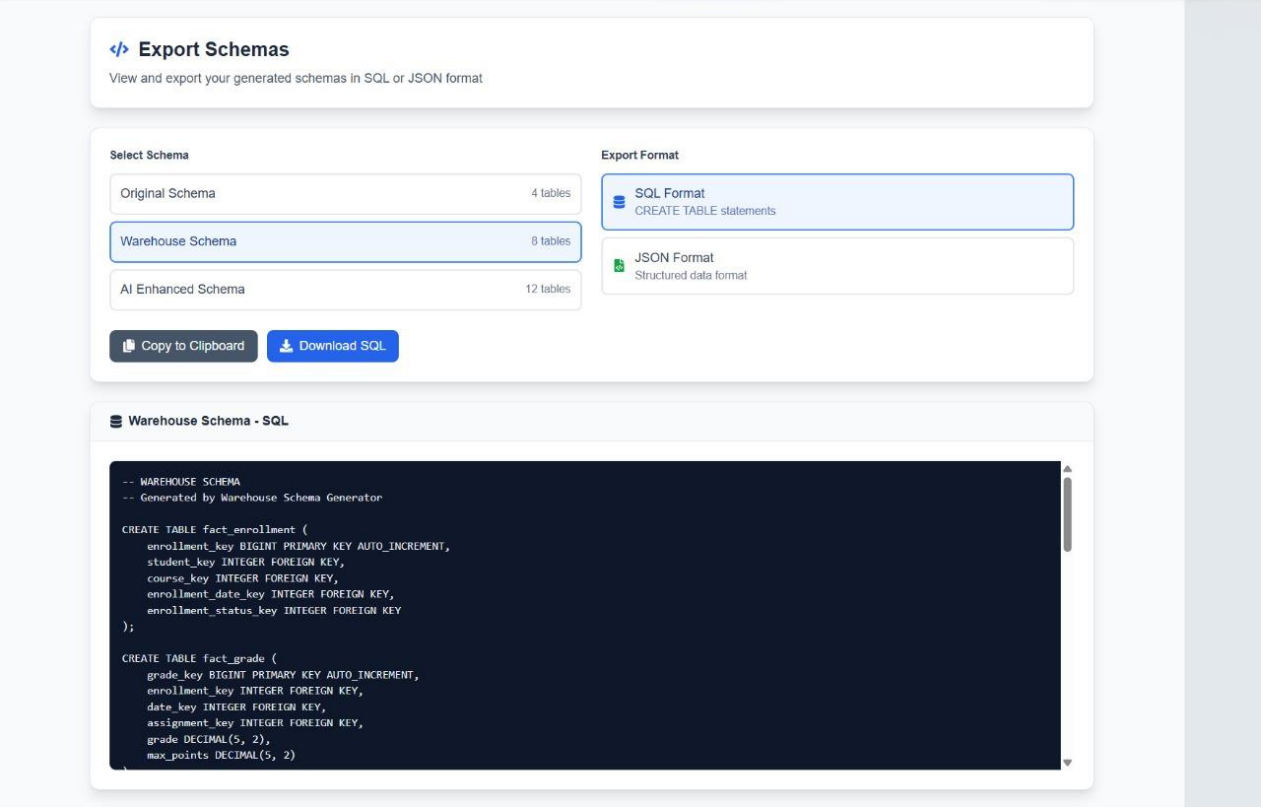

Figure 5.15: Download Schema Report

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

The experiments and evaluations conducted in Chapter Four demonstrate that DataForge effectively automates schema design, achieving a balance of speed, accuracy, and alignment with both technical and business requirements. The hybrid SQL-AST and regex parser achieved a parsing accuracy of 96% across diverse schemas, reliably extracting tables, columns, and constraints. Schema generation was robust, with heuristic classification combined with AI enhancements producing star schemas in under 5 seconds for 100-table schemas, meeting performance targets and surpassing prior cost-based approaches. Structural and semantic fidelity were maintained, with 100% structural similarity in table counts and relationships, while names and groupings improved by 5.1%, enhancing analyst comprehension. AI-driven suggestions increased Kimball-style compliance, including support for slowly changing dimensions, surrogate keys, and audit fields, from 70% to 75.6%, highlighting the value of domain-aware enhancements. Schema quality and integrity also improved, with the Schema Quality Index rising from 94.7% to 97.6%, indicating more complete, consistent, and concise schemas. Relationship integrity improved from 85.9% to 87.8%, and domain alignment rose from 88.4% to 95%, reflecting stronger adherence to business rules. User Acceptance Testing with ten participants yielded a mean satisfaction score of 4.2 out of 5.0, with particular praise for real-time validation and interactive editing features. These results confirm DataForge's ability to deliver high-quality, usable schemas that meet the needs of data analytics teams.

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 extend **Dataforge's** capabilities, we propose two key directions for future development. These recommendations build on the current system while exploring ambitious and distinct enhancements to ensure long-term innovation in automated data warehouse schema

1.Template Management and Sharing

To enhance adaptability and collaboration, DataForge could introduce:

- A user-driven template editor for custom schema designs.
- A community repository for sharing industry-standard templates.

Benefits of template editor:

- Enable users to create and save tailored dimension and fact table templates (e.g., retail-specific promotion dimensions or healthcare compliance tables).
- Support customization for unique business needs, improving schema relevance.
- Provide a user-friendly interface for defining table structures and relationships.

Community repository features:

- Host a shared library of best-practice patterns across industries.
- Allow users to contribute, access, and version-control templates with metadata for domain relevance and usage statistics.
- Facilitate cross-organization knowledge sharing for rapid schema design.

Technical needs:

- Develop a frontend template management interface integrated with the backend for storage and retrieval in a standardized format.
- Implement version control to track template updates and ensure consistency.

Impact:

- Enhance DataForge's flexibility for niche domains.
- Foster collaborative data engineering practices through shared resources.
- Accelerate schema design by leveraging community-driven standards.

2. Extensible AI Modules

To enhance flexibility and user trust, DataForge could introduce:

- A plugin architecture for custom AI modules.
- Explainable AI (XAI) for schema suggestions.

Benefits of plugin system:

- Allow integration of proprietary rule engines or ML models.
- Customize logic for specific domains (e.g., healthcare compliance, retail forecasting).
- APIs to let external modules interact with DataForge's parsing and generation pipeline.

Explainable AI features:

- Justify AI-generated suggestions (e.g., why a Promotion_Dim was added).

- Use techniques such as:
    - Feature importance scores
    - Natural language explanations
    - Frameworks like SHAP or LIME

Technical needs:

- Extend current AI pipeline (TF-IDF, BERT, LLM) with explainability modules.

Impact:

- Improve transparency and customization.
- Boost user confidence and adoption in specialized industries.
- Advance research in explainable AI for data engineering

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**