ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
CESENA'S CAMPUS

Engineering and Architecture's school
Master Degree Course in Engineering and Computer Science

Evaluating Large Language Models for Dimensional Fact Model Design with Automated Pipelines

Thesis in
Business Intelligence

Thesis supervisor
Prof. Enrico Gallinucci

Candidate
Luca Rubboli

2nd Graduation session
Academic Year 2024-2025

Abstract

This work investigates the use of large language models for conceptual design of multidimensional data warehouses, comparing supply-driven and demand-driven approaches. In the supply-driven approach, Dimensional Fact Model schemata is generated from source relational schemas, whereas in the demand-driven approach, schemata is generated from textual end-user requirements. Multiple LLMs are evaluated, including GPT, LLaMA, Falcon and Mistral, using automated pipelines for YAML-based schema extraction, metrics computation and visualization. Evaluation metrics include node- and edge-level precision, recall and F1-score, as well as custom error metrics reflecting domain-specific schema errors. Experiments are run on CPU and GPU environments, with automated scripts ensuring reproducibility and consistent execution across multiple runs. Results show that prompt engineering significantly improves model performance: for supply-driven design, average F1-scores nearly double, while for demand-driven design, careful prompt design increases scores by up to 20%. GPT-5 demonstrates slight improvements over GPT-4, particularly in capturing relational dependencies. The study also highlights practical limitations, including memory constraints with larger import models, variability in execution times and the need for manual post-processing rules. Future work includes expanding the exercise dataset, developing automated alignment strategies, exploring interactive multi-turn schema design and experimenting with fine-tuning large import models to enhance both accuracy and efficiency. These results provide a systematic foundation for leveraging LLMs in automated data warehouse conceptual design, balancing effectiveness and computational resources.

I wish to dedicate this work, and the entire journey, to all those who have been by my side with patience and affection.

Acknowledgements

I'm eager to express my sincere gratitude to all the professors I faced throughout my studies. In particular, I thank my supervisor Dott. Prof. Enrico Gallinucci, for his guidance and feedbacks, as well as availability and kindness, but also the laboratory staff and teaching assistants who provided invaluable help in practical sessions and research: your help has been fundamental.

I would like to thanks for all working experiences I had during these years, with specific regard to teaching activities in primary schools, for all that have taught and given to me.

I also want to thank my friends and colleagues I met during these years at university. What a journey, a wonderful journey we had. Your support, encouragement, help and shared experiences have made this years so memorable and enjoyable. Without you all this wouldn't have been the same; I would like to remark that even if this can be seen as an end to those wonderful days and nights we lived together, we all should focus on how much effort we put in pushing every moment up to the limits, without regrets by my side, hoping the same for yours; but even if it looks like an end is put only for good things, the same happens for the bad ones too: every moment it was your turn to pay for the coffe or everybody pretends to not have pennies, when you felt mocked by someone who outperformed you in shortcut knowledge, but also when a bathroom door decided to imprison you for hours generating gags. You're too many to list as singles, Marajas, conspiracy theorists, Abruzzo's citizens, board gamers and also ones who made me doubt about bias accumulated during my life. I wish you all the bests and a brightful future, and in case everything goes wrong, a Maraffa or Cervellone player professional career will be a second-best option for sure.

An honorable mention goes to my longtime friends, who have seen a lot of changes in me, but have nevertheless stood by my side with patience and affection. I'm aware that those nights at the bar or clubs look so remote, but I remain grateful for every laugh and moment we shared, and even more for the fact that you're still here now. I truly hope that in the future we'll be more in touch and continue building new memories together.

Of course, I would love to thank my girlfriend for the support, the patience and perseverance in detaching from work to remember the importance of truly living and embracing experiences of all kinds, from extreme ice driving to silly falls on skis. I'm really grateful for all experiences we did together and for the ones to come.

Last, though certainly not least, my family, for believing in me and for your unconditional support, which has been fundamental for finding the inner strength to persevere even in the toughest moments and for convincing me to take the leap six years ago looking for my life path, starting from university. The news of becoming uncle became a source of inspiration, driving me to complete this journey at my best.

As a new Doctor, and above all, as a person, a huge and heartfelt thank you!

Contents

List of Figures

Listings

Chapter 1

Introduction

The rapid growth of digital technologies has led to unprecedented availability of data in almost every domain of science, engineering and business. Organizations increasingly rely on these data not only to monitor ongoing processes, but also to support strategic decisions, optimize operations and forecast future trends. To transform raw data into actionable knowledge, structured approaches to data organization and analysis have become essential.

Within this context, data warehouses play a crucial role providing a unified, consistent and historical view of organizational data. They serve as the foundation for Business Intelligence systems, enabling reporting, analytics and advanced decision support. The design of a data warehouse, however, is a complex process that requires bridging the gap between business requirements and technical implementation. At the core of this process lies conceptual modeling, which provides a high-level representation of the analytical needs of stakeholders and guides subsequent logical and physical design steps.

Among the conceptual models proposed, the Dimensional Fact Model stands out as an intuitive and expressive framework. By structuring data around facts, measures, dimensions and hierarchies, the DFM facilitates communication between designers and decision-makers while ensuring a solid foundation for multidimensional analysis.

Despite its strengths, the design of conceptual models remains a demanding activity, often requiring significant expertise and extensive interaction with domain experts. This raises the question of whether emerging Artificial Intelligence technologies, and in particular Large Language Models, can assist in the modeling process. LLMs have demonstrated remarkable capabilities in understanding natural language, reasoning over complex information and generating structured outputs: properties that make them promising candidates for supporting both supply-driven and demand-driven design approaches in data warehousing.

The aim of this thesis is to analyze the potential of Large Language Models to assist in the design of data marts, considering both supply-driven and demand-driven methodologies.

Thesis Structure. The structure of this thesis is as follows. Chapter 2 provides the theoretical foundations for this work. It introduces the core concepts of data warehousing, including OLAP and OLTP systems, ETL processes and conceptual models for data marts. It also presents the Dimensional Fact Model as a central framework for conceptual modeling in Business Intelligence, detailing its basic and advanced constructs. In addition, covers fundamental concepts in Artificial Intelligence, with a focus on Natural Language Processing and reviews relevant architectures and Large Language Models that will be analyzed for their potential in supporting both supply-driven and demand-driven design approaches. Chapter 3 presents a detailed definition of the research problem, while Chapter 4 outlines design of the proposed application, serving as the bridge between the theoretical foundations introduced in the background chapter and the practical experimentation and analysis that follow. The subsequent chapters describe the methodology, experiments, results and analysis of LLM capabilities in assisting DFM-based conceptual modeling. Chapter 5 presents the implementation details of the proposed system, including design of the evaluation pipeline, object-oriented abstraction for model handling, post-processing framework, as well as aggregation and visualization components. Chapter 6 reports the experimental setup, the prompts and models tested and provides a comprehensive analysis of results obtained under both supply-driven and demand-driven approaches, with detailed evaluation metrics and error analyses. Finally, Chapter 7 summarizes the key contributions of this work, discusses its limitations and outlines future directions for research on leveraging LLMs for conceptual modeling in data warehousing.

Chapter 2

Background

2.1 Data Warehousing

The concept of data warehousing emerged in the late 1980s as organizations increasingly recognized the need to consolidate and analyze large amounts of data generated by their heterogeneous information systems. Traditional operational databases were designed primarily to support standard business activities, but were not well suited for strategic decision-making processes that required the integration, cleaning and aggregation of data from multiple sources.
Data warehousing is the process of collecting, integrating, storing and managing large volumes of data from heterogeneous sources in a centralized repository, the **Data Warehouse**, with the purpose of supporting reporting, analysis and decision-making, using formats optimized for analytical queries.
Data warehouses play a central role in modern organizations by serving as the foundation for **Business Intelligence** and advanced analytics: analytical tools allow decision makers to explore the data, derive insights and support strategic choices. Despite their advantages, the design and maintenance of data warehouses remain complex tasks: designers must translate <u>evolving business requirements</u> into formal structures while ensuring consistency and scalability.
Moreover, another difficulty lies in the semantic gap between stakeholders, who express needs in natural language, and designers, who must formalize needs into precise models. This gap often leads to iterative refinements, higher costs and risks of misalignment between the implemented system and actual analytical needs.

2.1.1 Data Warehouse

A Data Warehouse is a collection of data designed to support the decision-making process, characterized by the following properties:

- **Subject-oriented**: it is organized around key subjects of the organization, rather than applications. The main focus is on business processes and decision-making needs, rather than being constrained by the operational structure of transactional systems;

- **Integrated and consistent**: given the heterogeneous sources as input data, integration phase involves reconciling differences in naming conventions, data types and formats. Consistency ensures that the data is reliable, unambiguous and conforms to defined rules, enabling accurate cross-source analyses;

- **Time oriented**: stores historical snapshots of data to enable trend analysis and time-based comparisons so that analysts can observe changes, identify patterns and make informed strategic decisions;

- **Non-volatile**: loaded data is read-only for analytical purposes, stable and not altered by operational updates.

A remarkable difference in this field is between **Online Transaction Processing** (OLTP) systems and **Online Analytical Processing** (OLAP) systems. OLTP systems are designed to support database operations of organizations, such as order management, banking transactions, or inventory updates. Their focus is on reliability, speed and consistency for a large number of short and even concurrent transactions. OLAP systems, in contrast, are designed to support decision-making and strategic analysis: OLAP data are primarily read-oriented and updated only through controlled batch processes, which guarantees that analytical queries are always executed on a stable and consistent dataset, unaffected by the frequent changes that occur in operational databases. As a result, historical data are preserved in an immutable form, enabling long-term trend analysis, comparison across different time periods and reproducibility of analytical results. While OLTP ensures the smooth functioning of daily business processes, OLAP provides the insights necessary to guide long-term planning and performance evaluation.

2.1.2 Architectures

The architecture of a data warehouse defines how data is acquired, stored and accessed to support decision-making. A well-designed architecture must satisfy several fundamental requirements to ensure effectiveness and long-term sustainability:

- Separation: analytical and transactional processing must be kept as separate as possible. This avoids conflicts between day-to-day operations and analytical workloads, ensuring performance and reliability for both;

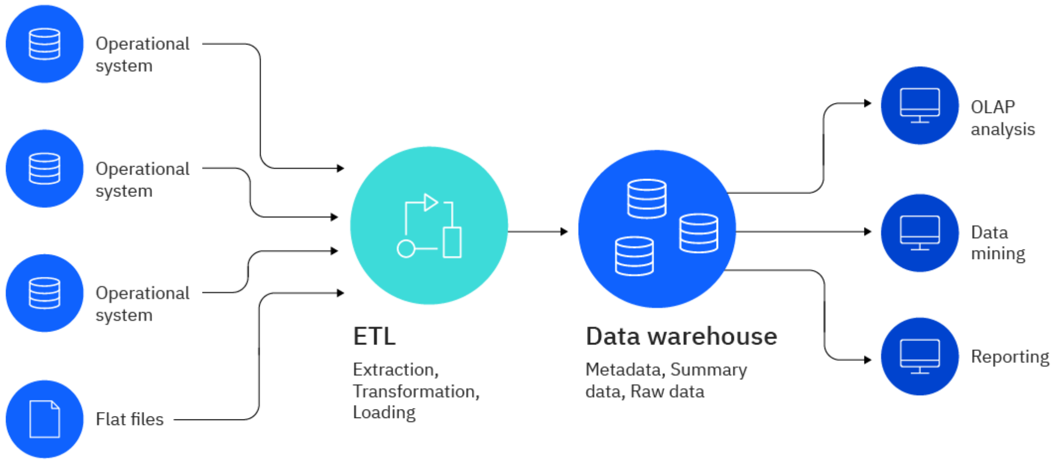


Figure 2.1: Data Warehousing overview
Source: `https://www.ibm.com/think/topics/data-warehouse`

- Scalability: hardware and software infrastructures should be easily scalable to handle the inevitable growth of data volumes and the increasing number of users over time;

- Extensibility: system should support new applications, analytical tools and emerging technologies without requiring a complete re-design;

- Security: access control mechanisms are crucial, given the strategic and often sensitive nature of stored data;

- Manageability: administrative complexity should remain manageable. Maintenance, monitoring and tuning should not become excessive burdens for IT teams.

Figure 2.1 provides a high-level overview of the typical architecture. Data from heterogeneous sources are first extracted, transformed and integrated through the **ETL** process. This integrated data is then stored in the data warehouse, where it is organized according to multidimensional models.

The architecture of a data warehouse can be organized into different levels, depending on how data storage, processing and access layers are separated. Here is an overview on the most common approaches.

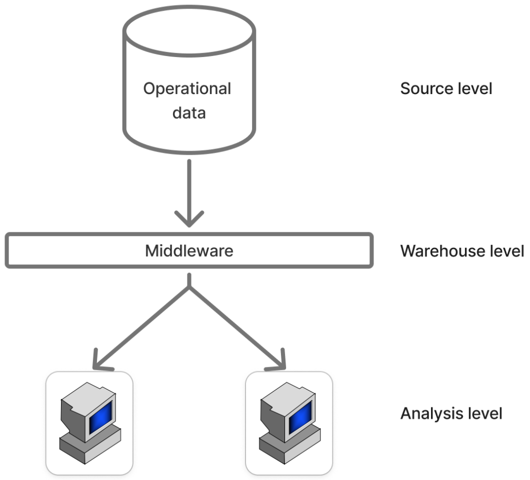


Figure 2.2: 1-level architecture.

1-Tier Architecture

In this configuration, the data warehouse resides directly on the database layer. Analytical tools access the same system that stores the data. Advantages: Simplicity, reduced latency and lower cost. Limitations: Lack of separation between data storage and analysis workloads, which often results in performance bottlenecks, especially handling both OLTP and OLAP operations.

2-Tier Architecture

Physical database layer separates from the analytical tools. The data warehouse is stored in a dedicated database and client applications query the warehouse directly. Typically, data warehouse is splitted in **data marts**: specialized subset designed to serve the analytical needs of a specific business unit, department, or function within an organization. Unlike the enterprise data warehouse, which integrates and consolidates information from across the entire organization, it's typically more focused on data relevant for a particular group of users. Advantages: Clearer separation of concerns, better query performance compared to the 1-tier model, high-quality information continuously available at the warehouse level.

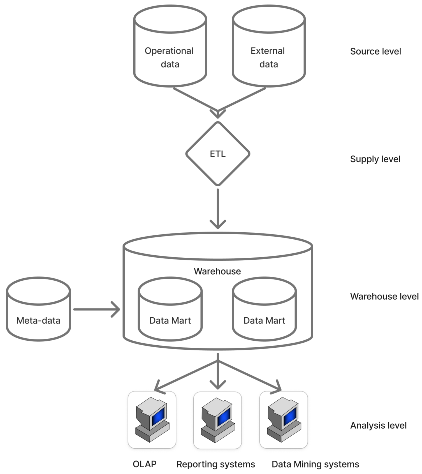


Figure 2.3: 2-levels architecture.

Limitations: Limited scalability, as the analytical layer and the warehouse are still closely connected, which may create problems with large or complex workloads.

3-Tier Architecture

Introduces an intermediate layer, the **Operational Data Store**: database designed to integrate data from multiple operational systems in near real-time, providing a consolidated and consistent view of transactional information. In contrast with data warehouse, the ODS focuses on supporting day-to-day operations and short-term decision-making by offering up-to-date and detailed data. ODS acts as an intermediate layer between operational systems and data warehouse, enabling data cleaning, reconciliation and transformation before the information is loaded into the warehouse. Advantages: High scalability, flexibility and efficiency for analytical queries, improves data quality; well-suited to large enterprises. Limitations: More complex to implement and manage, requiring advanced infrastructure and administration.

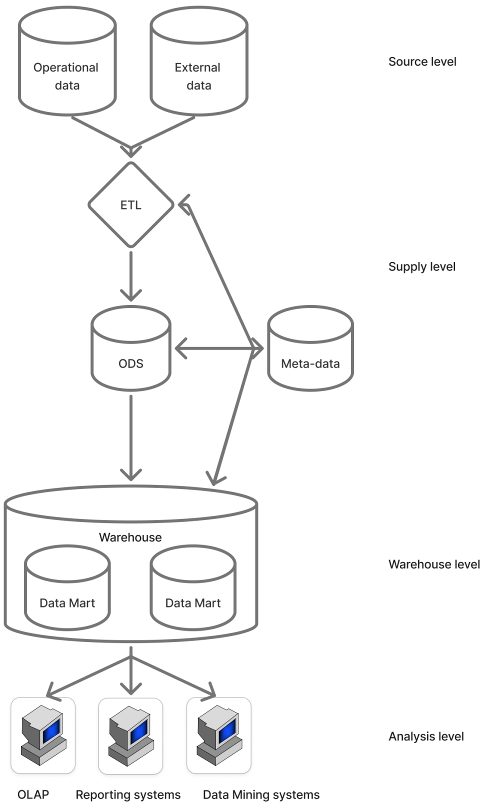


Figure 2.4: 3-levels architecture.

2.1.3 ETL

The ETL pipeline, which stands for Extract, Transform and Load, integrates data from heterogeneous sources. Its role is to feed a single, detailed and high-quality data source that populate the Data Warehouse. In the extraction phase, data are collected from operational systems and external sources. During transformation, data is cleaned, standardized and reconciled in order to resolve inconsistencies and ensure integration. Finally, in the loading phase, transformed data is stored in the warehouse, where it becomes available for analytical processing.
The ETL process thus serves as the bridge between transactional systems (OLTP) and analytical environments (OLAP), ensuring that decision makers can rely on complete, consistent and up-to-date information.

Extraction

In this phase, relevant data are extracted from the sources. 2 kind of extractions may be identified.

- **Static** extraction is performed when the Data Warehouse must be populated for the first time and conceptually corresponds to a snapshot of the operational data;
- **Incremental** extraction is used for periodic updates of the warehouse and captures only the changes that have occurred in the sources since the last extraction. This can be implemented by means of logs maintained by the operational DBMS, time-stamps or source-driven approaches.

The choice of which data to extract depends primarily on their quality.

Transformation and Cleaning

The main goal of this phase is to improve the quality of source data, addressing issues such as duplicated data, inconsistencies between logically related (or due to mere typographical errors) values, missing data, improper use of a field, impossible or incorrect values After cleaning, data from the operational source format are converted into the one required by the warehouse. The correspondence with the source level is complicated by the presence of heterogeneous and distinct sources, which requires a complex integration process, Reconciled data follow a pipeline of conversion and normalization, operating on formats and units of measure to standardize the data, then matching, establishing correspondences between equivalent fields across different sources and finally selection, reducing the number of fields

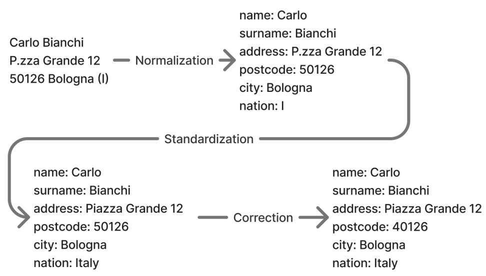


Figure 2.5: ETL cleaning & transformation example.

and records compared to the sources. The process of feeding the Data Warehouse involves denormalization, in contrast with normalization previously enrolled, to improve query performance and to align with multidimensional modeling. Moreover, aggregation is introduced, producing the appropriate summaries of the data.

Loading

The loading of data into the warehouse can be carried out in two ways:

- Refresh: the DW is completely rewritten, replacing all previous information. Typically used during the initial population;
- Update: only changes occurred in the source data are added to the DW. Commonly used for periodic updates of the DW.

2.1.4 Life cycle

A Data Warehouse is not a static system but **evolves through a life cycle** that involves design, implementation, maintenance and continuous adaptation to business needs. The life cycle includes:

Requirements analysis – identification of business goals, analytical needs and data sources.

Conceptual and logical design – definition of multidimensional models (facts, measures, dimensions, hierarchies).

Physical design and implementation – schema creation, ETL pipelines, storage and indexing.

Deployment and usage – the system is populated and accessed by BI tools and end users.

Maintenance and evolution – periodic updates, performance tuning and adaptation to new business requirements.

A crucial aspect in the development phase is the choice of design methodology

Top-Down Approach

Proposed in [8], begins with an enterprise-wide data warehouse at a high level of abstraction. Data marts, usually smaller, are derived later.

Strengths:

- Provides a unified and consistent data model across the organization, with a global vision of the objectives;
- Facilitates integration and reconciliation of heterogeneous data sources;
- Better suited for long-term scalability.

Weaknesses:

- High costs and complexity at the beginning;
- Longer time to validate project, as benefits are visible only after significant development, which reduces both interest and trust in the initiative.

Bottom-Up Approach

Popularized in [10], builds the data warehouse incrementally, starting with independent data marts built for specific departments or analytical needs, later integrated into a broader warehouse.

Strengths:

- Faster deployment and shorter time-to-value, resulting in fast feedbacks on the usefulness of the system under development;
- Focused on specific business requirements from the start, simplifying analysis and implementation;
- Lower initial cost.

Weaknesses:

- Risk of inconsistency across different data marts, due to partial view of the overall business domain;

- Integration into a unified warehouse can become complex later;
- May require re-engineering if organizational needs grow.

2.2 Data Mart design approaches

When designing a data mart, two main strategies can be considered: supply-driven and demand-driven. Both approaches aim to determine which data to include and how to structure them, but differ in design starting point.

2.2.1 Supply-driven approach

In data-driven approaches, data marts are designed starting from a detailed analysis of the operational sources. User requirements still play a role, but primarily in guiding the designer to select the relevant portions of data for decision-making and to structure them according to the multidimensional model.

Strengths:

- High-level conceptual schema for data marts can be algorithmically derived from the reconciled data level, based on the structure of the sources;
- Design of the ETL process is greatly simplified, since each item of information in the data mart is directly associated with one or more attributes of the sources.

Weaknesses:

- User requirements are assigned only a secondary role in determining the informational content for analysis;
- Designer receives limited support in identifying facts, dimensions and measures.

This approach is suitable in situations where in-depth knowledge of the sources feeding the data mart is either available or can be obtained at reasonable cost and time, when source schemas are well normalized and complexity is not excessive. These conditions are typically satisfied when the chosen architecture includes a reconciled data level, which ensures normalization and deep knowledge of the sources. The same applies in cases where the source is a single, well-designed, relatively small database. Design experience shows that, whenever applicable, the data-driven approach is generally preferable to other methods, as it **allows to achieve a high-quality conceptual schema more efficiently**.

2.2.2 Demand-driven approach

In demand-driven approaches, the design starts with determining the informational requirements of the data mart users. The challenge of mapping these requirements to the available data sources is addressed only afterward, through the implementation of appropriate ETL procedures.

Strengths:

- User needs are placed at the forefront of the design process.

Weaknesses:

- Designers must exert considerable effort during the design of the data loading process;
- Facts, measures and hierarchies are derived directly from user specifications, and only afterward can be verified whether the requested information is actually available in the operational databases;
- The user's trust in both the designer and the usefulness of the data mart may be compromised if expectations are not met.

Given that it is generally more difficult to pursue than the data-driven approach, it is often the only viable option when a detailed prior analysis of the sources is not feasible (e.g. when data mart is fed by an ERP system), or when sources consist of complex legacy systems, because of complex normalization.

2.3 Conceptual Models for Data Warehousing

A crucial step in the design of a data warehouse is the conceptual modeling phase, where business requirements are abstracted into a schema that captures the essential facts and dimensions of analysis. Several approaches have been proposed in the literature, each with a different balance between <u>expressiveness</u>, <u>formality</u> and <u>ease of communication</u>. The choice of model often depends on the complexity of the domain, the familiarity of stakeholders with modeling notations and the degree of alignment required between business and technical perspectives.

Here is a concise overview of the most common conceptual models with comparison in Table 2.1.

- **Entity-Relationship** extensions: traditional ER models have been adapted for analytical contexts by introducing multidimensional extensions (e.g. facts, dimensions, hierarchies). They provide formal rigor and are often familiar to database designers, but may be less intuitive for non-technical stakeholders, introduced in [19];

- **Dimensional Fact Model**: graphical high-level model, detailed in [6], designed specifically for multidimensional analysis, that represents information in terms of facts, associated with measures and dimensions including hierarchies that allow users to explore data at different levels of granularity. Requires familiarity with specific notation.

Table 2.1: Comparison of conceptual modeling approaches in Data Warehousing.

Model	Strengths	Weaknesses
Entity–Relationship Extensions	Well-known to database designers Formally rigorous	Less intuitive for business users Limited multidimensional support
Dimensional Fact Model (DFM)	Graphical and intuitive Effective for communication Captures facts, measures, hierarchies	Requires specific notation knowledge

2.3.1 Dimensional Fact Model

The Dimensional Fact Model (DFM) is a graphical conceptual model for data marts, designed to support the conceptual design process, provide an environment in which **formulate queries intuitively**, as well as facilitate the dialogue between designer and end user to refine requirement specifications, establish a stable platform as base for logical design but still independent from the choice and deliver clear and unambiguous documentation after implementation. The conceptual representation generated by the DFM consists of a set of fact schemas. The fundamental elements modeled within fact schemas are:

- *Fact*: concept of interest for the decision-making process. Models a set of events that occur within the business (e.g. sales). The key characteristic is its dynamic nature, meaning evolvution over time;

- *Measure*: numerical property of a fact, describing a quantitative aspect relevant for analysis;

- *Dimension*: property with a finite domain associated with a fact, providing a coordinate of analysis;

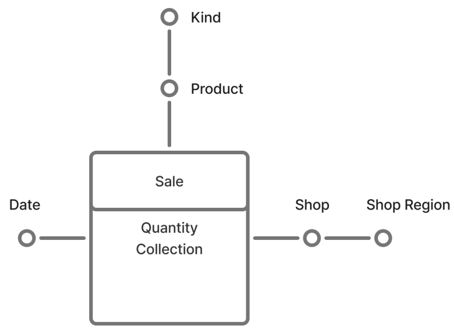


Figure 2.6: DFM example.

- *Hierarchy*: directed tree whose nodes represent dimensional attributes and whose edges model many-to-one associations between pairs of attributes;
- *Dimensional attribute*: refers both to dimensions and to other discrete-valued attributes that describe them.

Events

A **primary event** is a specific occurrence of a fact, identified by a tuple consisting of one value for each dimension. Each primary event is associated with a value for every measure. A **secondary event** instead, is an aggregated occurrence of a fact, obtained by grouping primary events according to a set of dimensional attributes, so-called the group-by set. It represents a higher-level view of the data, where each measure is summarized across the underlying primary events. Thus, hierarchies define how primary events can be meaningfully aggregated and selected for decision-making. The dimension at the root of a hierarchy determines the finest level of aggregation granularity, while the other dimensional attributes correspond to progressively coarser levels of granularity.

Advanced Constructs

While the basic building blocks of a Dimensional Fact Model provide a solid foundation for conceptualizing a data mart, real-world applications often require more sophisticated constructs to capture complex relationships and nuances in the data for a more expressive and flexible representation.

- **Descriptive attribute**: provides additional information about a dimensional attribute within a hierarchy, to which it is connected through a one-to-one association. It is not used for aggregation, since its values are typically continuous and results from a one-to-one relationship;
- Some edges in the fact schema may be **optional**;
- **Shared hierarchy**: shorthand used to denote that a portion of a hierarchy is replicated multiple times within the schema;
- **Convergence**: situation in which two dimensional attributes can be connected by two or more distinct directed paths, provided that each of them still represents a functional dependency;
- **Cross-dimensional attribute**: dimensional or descriptive attribute whose value is determined by the combination of two or more dimensional attributes, which may belong to distinct hierarchies;
- **Multiple edge**: models a many-to-many association between two dimensional attribute;
- **Incomplete hierarchy**: hierarchy in which, for some instances, one or more aggregation levels are missing (either because they are unknown or undefined);
- **Recursive hierarchies**: the parent-child relationships between levels are consistent, but instances may have different lengths.

Aggregation and Additivity

Since hierarchies define how primary events can be aggregated into secondary events, it becomes essential to specify how measures behave under such aggregation. This requires defining suitable operators that combine the values of measures from primary events into new values associated with secondary events. From this perspective, measures can be classified into categories:

- **Flow** measure: refers to a period of time and is evaluated cumulatively at its end (e.g. the number of products sold in a day);
- **Level** measure: evaluated at specific points in time (e.g. the number of products in stock).

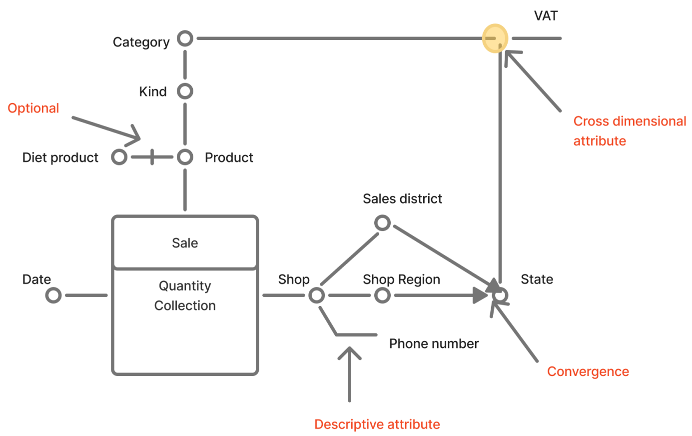


Figure 2.7: DFM example with advanced constructs.

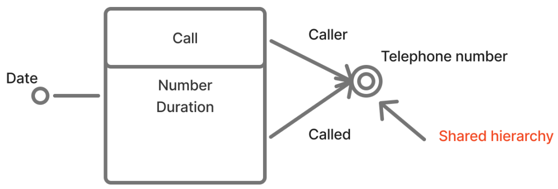


Figure 2.8: DFM example with advanced construct.

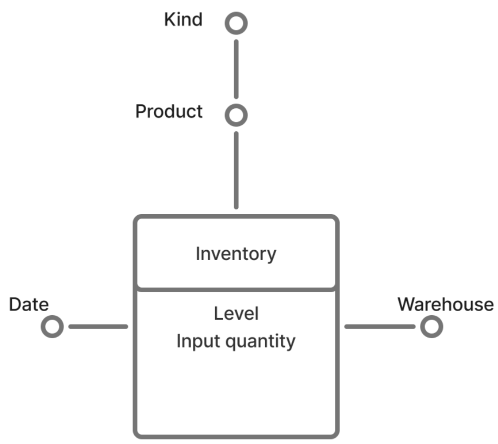


Figure 2.9: DFM example for aggregation analysis.

- **Unit** measures: evaluated at specific points in time too, but expressed in relative terms (e.g. the unit price of a product).

A measure is additive with respect to a dimension if its values can be aggregated along the corresponding hierarchy using the sum operator. Otherwise, it is considered non-additive. A non-additive measure is non-aggregable if no aggregation operator can be meaningfully applied to it.

	Level	Input quantity
Date	AVG,MIN	SUM
Product	SUM	SUM
Warehouse	SUM	SUM

Table 2.2: Analysis of operators for measures aggregation belonging to DFM in Figure 2.9

2.4 Artificial Intelligence

Artificial Intelligence is a broad field of computer science dedicated to creating systems capable of performing tasks that typically require human intelligence, including reasoning, learning from experience, problem solving and natural language understanding. Although the term was first formally introduced in the mid-1950s, AI has evolved through multiple paradigms, from symbolic approaches and rule-based systems to machine learning, up to deep learning.

Today, AI technologies are embedded in a wide range of applications, including autonomous systems, computer vision, speech recognition, natural language processing and decision support systems. The common characteristic of these applications is their ability to process complex data and adapt their behavior based on patterns and knowledge extracted from **experience**. The accuracy of this knowledge depends both on the **amount of information** available and on the **quality of the data** in modeling problems as precisely as possible.
A key component of modern AI is **Machine Learning**, which provides computational methods that allow systems to learn patterns from data without being explicitly programmed for every task. Algorithms improve their performance as they are exposed to more data, identifying correlations, making predictions and discovering insights that would be difficult to encode manually. This ability to generalize from examples makes it a fundamental tool in applications ranging from predictive analytics to recommendation systems and autonomous decision making.

2.4.1 Training

The discipline of Machine Learning focuses on extracting knowledge models from datasets in order to generate accurate predictions. These models are composed of parameters, whose number is proportional to the complexity of the problem and the desired level of accuracy. Some parameters need to be adapted to the problem, which is why a **training phase** is generally adopted, while others, called hyperparameters, are defined by the programmer during the model's creation.

Two primary paradigms are commonly distinguished: *supervised* and *unsupervised* learning.

Supervised learning relies on labeled datasets, where each input is paired with a corresponding output. The model learns a mapping between inputs and outputs, with the goal of making accurate predictions on unseen data. Typical applications include classification and regression.

Unsupervised learning, instead, operates on unlabeled data, seeking to uncover hidden structures or patterns without predefined outcomes. Its main tasks include clustering, which groups similar data points together and dimensionality reduction,

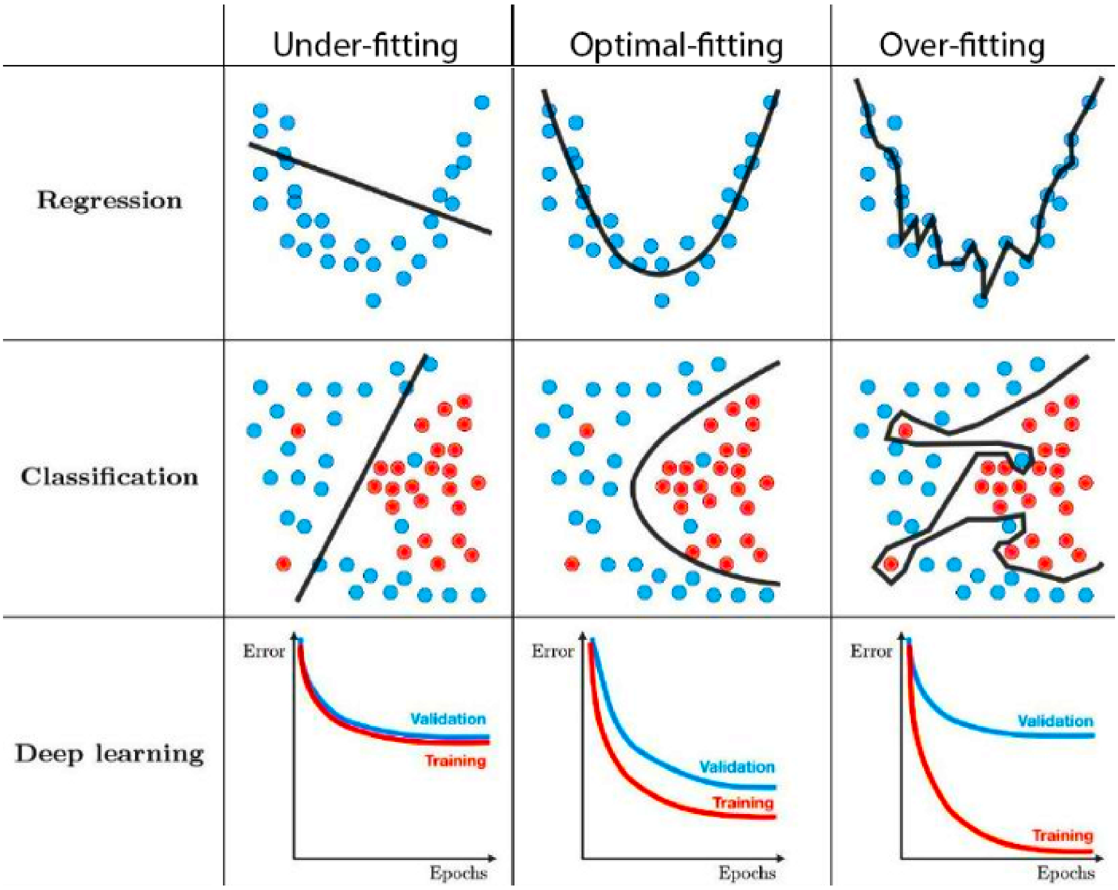


Figure 2.10: Examples of underfitting, optimal fitting and overfitting.
Source: `https://towardsdatascience.com`

which simplifies data while preserving its essential characteristics.

Together, these paradigms provide the foundation for many AI applications, ranging from predictive analytics to knowledge discovery.

To refine training process, it is common to split the dataset into three subsets: the **training set**, which is used to compute and fine-tune the parameters; the **validation set**, which estimates the model's generalization ability; and the **test set**, which allows a realistic evaluation of the model on unseen data. According to the **hold-out** method, the validation set helps determine whether a model is underperforming due to underfitting, poor accuracy on both the training and validation sets, or overfitting, where good results on the training set do not generalize to the validation set.

Underfitting can be caused either by the use of models that are too simple to capture the problem or by examples that do not correctly represent it. Overfitting, instead, indicates poor generalization to new data, which may result from an ex-

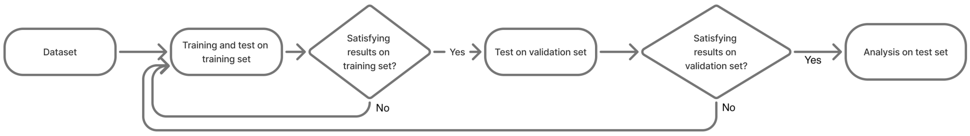


Figure 2.11: Workflow during training and validation phases.

cessively complex model that fits the training data too much or from an insufficient amount of data for proper training.

To mitigate these issues, it is necessary to:

- Filter the data, ensuring that irrelevant features are removed;
- Normalize data features onto a uniform scale to make the training phase more stable;
- Regularize parameters, constraining them to smaller values to simplify and generalize the model;
- Tune hyperparameters using cross-validation techniques in order to optimize the model's structure;
- Add new data to dataset.

During training, the objective is to reduce errors on the training set while maintaining the model's ability to generalize effectively to unseen data.

After obtaining good results on both training and validation sets, a further evaluation phase is performed using the test set, assessing model's performance and effectiveness.

In recent years, Artificial Intelligence has increasingly been integrated into the fields of Data Warehousing and Business Intelligence extending decision-making process by enabling systems to predict future trends and generate insights automatically. Moreover, Machine Learning techniques are employed to enhance data quality through anomaly detection, data cleaning and integration, but also to optimize query performance and uncover hidden patterns in large-scale datasets.

2.5 Natural Language Processing

Natural Language Processing is the discipline concerned with studying and developing methods for the processing of language, whether expressed in textual or

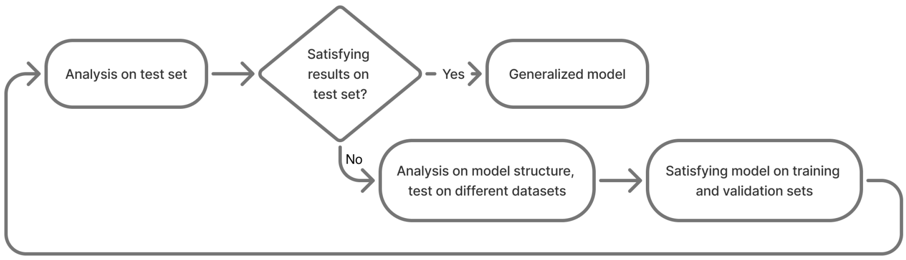


Figure 2.12: Workflow during test phase.

spoken form. The complexity of natural language processing arises from the diversity of languages, each with its own grammar rules, vocabulary and sometimes even distinct alphabets. A single language may also branch into dialects or linguistic variations and, depending on the context, the same expression can take on completely different meanings.

Additional layers of ambiguity are introduced by rhetorical figures such as irony and sarcasm, as well as by abbreviations and elements that transcend the language itself, such as emoticons.

The strong dependence on context in this problem makes the use of preprocessing techniques crucial for improving processing accuracy.

Natural language text is a form of **unstructured data**: preliminary processing is of crucial importance in order to normalize and contextualize the raw data.

For these purposes, various techniques have been developed:

- *Tokenization*: the division of a text into sentences and, subsequently, into words. For an accurate execution, language-specific models and rules are applied;
- *Part of Speech* tagging: the assignment, word by word, of the corresponding grammatical category;
- *Casefolding*: aims at eliminating ambiguities between uppercase and lowercase letters by converting text into a common standard;
- *Stopword removal*: the elimination of words such as articles that do not convey significant information; usually a predefined list is employed;

- *Lemmatization*: the replacement of each word with its lemma, i.e. its base form;
- *Stemming*: similar to lemmatization, it reduces each word to its morphological root. This generally requires a simpler algorithm, but can lead to information loss;
- *Word Sense Disambiguation*: the procedure that assigns the correct meaning to each word by identifying it within a specific context. Some algorithms, for instance, select the definition that shares the largest number of surrounding words;
- *Named Entity Recognition*: the identification of named entities referred to in the text, such as places, job roles, numbers, or dates. These elements are often highly relevant for contextualizing a given sentence.

2.5.1 Text representation

Another particularly important aspect concerns the representation of text, since machine learning models are not able to process data in textual form directly. Conversion is carried out through various techniques whose goal is to obtain a compact and structured representation, including:

- *Bag of Words* (BoW) representation: a table is created with two columns, one listing the terms generally standardized through stemming to reduce the number of distinct words and the other recording their occurrences in the text. This technique is particularly effective in sentiment analysis, where the task is to classify text into categories. For instance, to classifying a review as positive or negative, the method relies on counting positive and negative words and assigning them weights;
- *Vector Space Model*: provides a representation of multiple documents following a Bag of Words metric based on the words of a shared dictionary. It contributes significantly to topic identification across documents. When integrated with *term frequency–inverse document frequency* (TF–IDF) schemes, it allows the relevance of a term in a document to be quantified. The term frequency corresponds to the number of occurrences in the text, while the inverse document frequency corresponds to the number of documents in the collection where the term appears. The latter acts inversely: the more documents a term appears in, the lower the weight assigned to it;
- *n-gram* representation: collects, at each iteration, sequences of words of length n. These sequences can identify entities or expressions composed of

multiple words, thereby enriching the Bag of Words structure previously described. The sliding window architecture of variable length is also significant as a probabilistic model for training in word generation tasks: given the preceding n–1 words, the goal is to predict the next word.

2.5.2 Word embeddings

The family of techniques for representing text as numerical vectors while preserving its semantics is referred to as word embeddings.

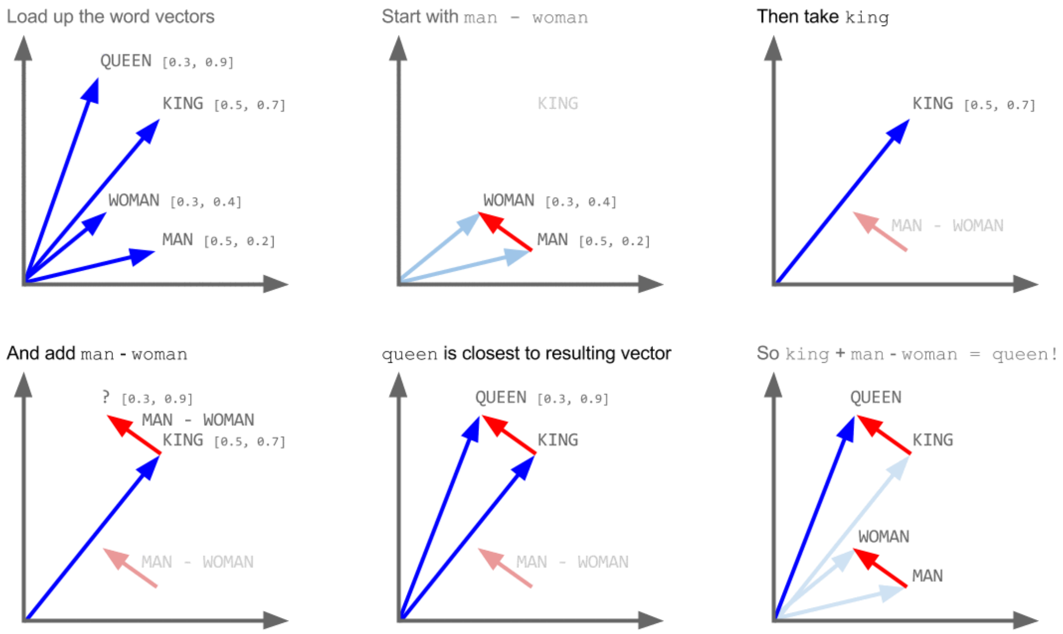


Figure 2.13: Terms projecting into a vector space based on semantics.
Source: https://multithreaded.stitchfix.com

These approaches are generally **unsupervised**, as the labels used to train the model are extracted directly from the data under analysis. The conversion is carried out through algorithms that project words into a vector space that retains semantic relationships:

- *One-Hot Encoding*: converts each word into a vector with a single value of 1, corresponding to its position in the analyzed document, and all other values set to 0. This allows for the creation of a distinct vector for every word in the document. However, it is poorly scalable with respect to document length, as it leads to a dimensionality explosion as vocabulary size increases, and it does not preserve semantic context;

- *Word2Vec* [13]: one of the most widely used embedding algorithms, consisting of two distinct training architectures. The Continuous Bag of Words (CBOW) model predicts a target word from its surrounding context (the preceding and following words), while the Skip-gram model predicts the context given a target word.

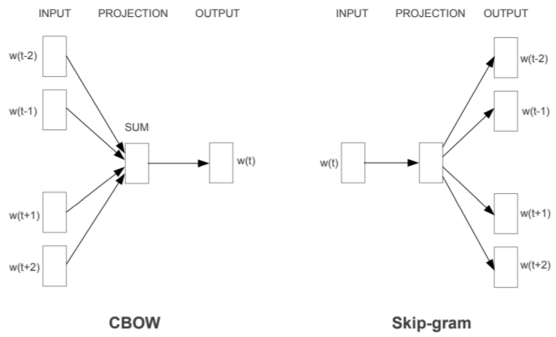


Figure 2.14: CBOW and Skip-gram architectures.
Source: https://towardsdatascience.com/nlp-101-word2vec-skip-gram-and-cbow-93512ee24314

The larger the dataset processed during training, the more accurate the resulting vector space will be, with words positioned according to a semantic similarity metric.

Other notable algorithms in this domain include GloVe [15], which constructs co-occurrence matrices between terms to derive accurate word representations, and Doc2Vec [13], which extends this principle to represent entire documents as numerical vectors, enabling similarity comparisons at the document level.

2.5.3 Sequence models

Sequence models are designed to process data where order matters, such as text, speech, or time series. Depending on how input and output sequences are handled, these models can be categorized as follows:

- *Sequence-to-Vector*: converts an input sequence into a fixed-length vector. Typical use cases include text classification or sentiment analysis, where the goal is to summarize the entire sequence into a single representation;

- *Vector-to-Sequence*: generates a sequence from a fixed-length input vector. This is common in generative tasks where a compact representation must be expanded into a structured output;

- *Sequence-to-Sequence*: transforms an input sequence into another sequence, possibly of different length. This architecture is widely used in machine translation, text summarization and speech-to-text applications. It often combines encoder–decoder structures with attention mechanisms for better handling of long-range dependencies.

These categories form the foundation of modern NLP architectures, including RNNs, CNN-based sequence models and transformer-based models. Each type offers trade-offs in terms of expressiveness, computational efficiency and ability to handle long range contexts.

2.5.4 Encoder - Decoder architecture

A model of this structure is composed of two main components:

- *Encoder*: a *sequence-to-vector* model that processes an input sequence and retains only the final output as a summary of the entire sequence;

- *Decoder*: a *vector-to-sequence* model that takes this summarized vector as input and generates an output sequence.

This sequence-to-sequence architecture takes an input sequence and produces a corresponding output sequence, making it suitable for various NLP tasks, such as machine translation, where the entire input sentence must be processed before any output can be generated.

However, this structure has a key limitation: by compressing the entire input into a single vector, the model's memory becomes constrained, making it difficult to accurately capture long range dependencies in longer sequences. This limitation motivated the development of the attention mechanism, which allows the decoder to selectively focus on relevant parts of the input at each generation step.

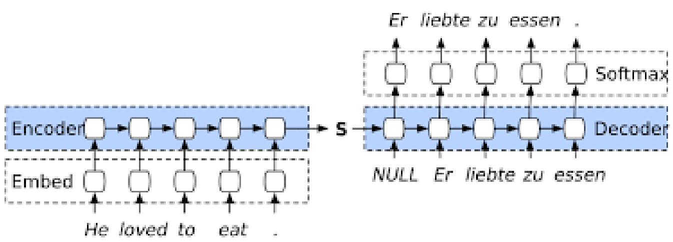


Figure 2.15: Encoder–Decoder architecture.
Source: https://towardsdatascience.com

2.5.5 Attention

The attention mechanism was originally introduced as an enhancement to the classical encoder–decoder structure, in order to overcome the limitations of the model's restricted memory and, consequently, its inability to handle long input sequences effectively.

The key idea behind attention is to allow the model to focus selectively on the most relevant parts of the input when generating each element of the output. Instead of relying solely on the encoder's final hidden state as a compressed representation of the entire input sequence, attention transforms the encoder into a *sequence-to-sequence* processor whose output is the full set of hidden states, each carrying contextualized information about different input positions.

At each decoding step, the decoder computes a weighted sum of these encoder outputs, where the weights indicate the relative importance of each input token for the current prediction. These weights are dynamically generated by the attention layer, which is trained jointly with the rest of the model. The training process aligns the decoder's output with the most relevant encoder states, effectively teaching the model where to focus when generating the next word.

This mechanism, introduced in [2], represented a breakthrough in neural machine translation and sequence modeling in general, as it significantly improved both accuracy and the ability to handle longer sequences.

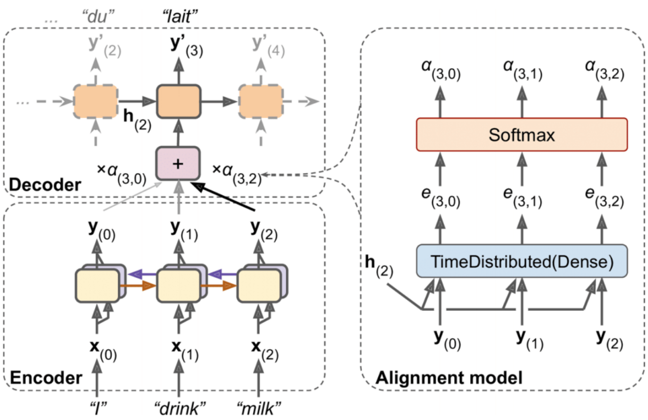


Figure 2.16: Attention mechanism integrated within the encoder–decoder architecture.

Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition

2.5.6 Transformers

The Transformer architecture, proposed in [20], marked a major advance in the state of the art for many Natural Language Processing tasks. Unlike previous

models, Transformers completely discard recurrent and convolutional layers, enabling faster and more parallelizable training, since input tokens no longer need to be processed sequentially.
At the core of the Transformer lies the **attention** mechanism, implemented within a deep encoder–decoder structure composed of multiple stacked layers. Each encoder layer consists of a *Multi-Head Attention* mechanism, allowing the model to capture multiple types of relationships and interactions between words in parallel, followed by a feed-forward neural network that processes and passes the output to the next layer.
To compute multi-head attention, each input embedding is projected into three distinct matrices: Query (Q), Key (K) and Value (V). These projections are learned during training. Attention scores are calculated by taking the dot product of a query vector with all key vectors, normalizing the result and then applying the scores to a weighted combination of the value vectors. This procedure enables the model to decide which words to emphasize when processing a given token.
Moreover, since the Transformer does not have recurrence or convolution to encode word order, it introduces positional embeddings that are added to the input embeddings. These embeddings inject information about the relative positions of tokens, allowing the model to preserve sequence structure.
During decoding, the encoder's final outputs are transformed into sets of keys and values that are passed to the decoder's attention layers. The decoder, in turn, applies masked attention to its own past outputs (to prevent looking ahead) and combines this with encoder–decoder attention to produce predictions. Finally, the output of the last decoder layer is passed through a linear projection into a high-dimensional vector, called the logits vector, from which a probability distribution is computed over the model's vocabulary to select the next output token.
This architecture has since become the foundation of modern LLMs, powering state of art models in translation, text generation, dialogue systems and beyond.

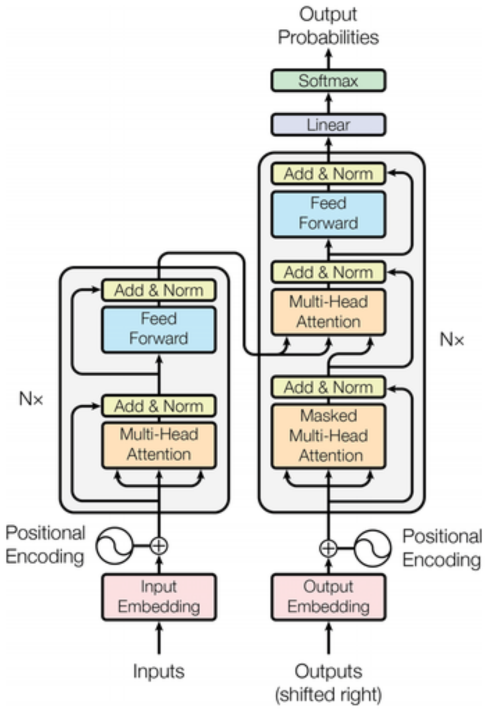


Figure 2.17: Transformer architecture.
Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition

2.6 Large Language Models

In recent years, Large Language Models have emerged as one of the most significant advances in Artificial Intelligence and Natural Language Processing. LLMs are built on the *transformer* architecture, which enables them to capture long-range dependencies in text and modelling contextual relationships more effectively than previous approaches such as recurrent neural networks or simple word embeddings. Their distinguishing feature is the massive scale of training, involving billions of parameters and vast amounts of textual data collected from diverse sources.

Unlike traditional NLP systems, often designed for a single specific task, LLMs adopt a general-purpose approach: through pretraining on large corpora and subsequent fine-tuning or prompting, they can adapt to a wide variety of applications without requiring task-specific architectures.

Prominent applications of LLMs include text generation, question answering, machine translation, summarization and dialogue systems. Lately, they are increasingly being integrated into domains such as software development and business intelligence for automated reporting and analytics.

The versatility of LLMs demonstrates their potential not only as tools for processing natural language but also as general reasoning engines that can assist in knowledge discovery and decision-making across a wide range of organizational and industrial contexts.

Main LLMs strengths:

- Advanced Language Understanding: LLMs can generate coherent, contextually appropriate text and perform tasks requiring nuanced comprehension;
- General-Purpose Functionality: Many LLMs can be adapted to multiple tasks through pretraining and fine-tuning or prompting, reducing the need for task-specific architectures;
- Automation of Complex Tasks: They can handle large-scale content creation, data analysis and knowledge extraction efficiently;
- Scalability: LLMs can process large volumes of data, supporting applications that require extensive textual reasoning.

Main LLMs weaknesses:

- Computationally Intensive: Training and deploying LLMs demands significant computational resources and energy;
- Bias and Fairness Concerns: LLM outputs can reflect biases present in the training data, leading to potentially unfair or undesirable results;
- No True Understanding: while LLMs generate meaningful text, they do not possess genuine comprehension or reasoning in a human sense;
- Data Dependency: Their accuracy and generalization heavily rely on the quality and diversity of training data.

LLMs can be integrated into applications in two primary ways: via **local import** or through **external APIs**.

When imported locally, typically from open source frameworks such as Hugging Face[1], the model weights and architecture are downloaded and executed directly on the user's hardware. In this setup, inference can be run on either CPU or GPU, depending on the available resources. GPUs offer significant acceleration for large-scale models, while CPUs may be sufficient for smaller ones or environments where

[1]HuggingFace source: `https://huggingface.co/`

GPU hardware is not available. Local imports provide full control over the model, enabling fine-tuning, customization, and offline operation. However, they require substantial computational resources and careful environment configuration.

By contrast, commercial providers such as OpenAI, make their models accessible via API endpoints. Computation occurs on remote servers, with users interacting through requests and receiving model outputs as responses. APIs provide ease of integration, scalability and access to cutting edge proprietary models, without the need for local infrastructure. The drawbacks include latency, per-request costs and data privacy concerns, since inputs are transmitted externally.

The decision between local import and API access depends on application requirements: APIs are often preferable for quick integration and state of the art performance, while local imports are better suited when customization, privacy, or hardware control is a priority.

Several LLMs, an overview is detailed in [22], have become prominent in recent years, each with distinct characteristics:

- GPT (OpenAI): exceptional in natural language understanding and generation, with multimodal capabilities and strong contextual reasoning. Drawbacks include high computational costs and occasional factual inaccuracies. A broader discussion of its capabilities can be found in the literature [7];

- Claude (Anthropic): designed for safety and alignment, capable of following complex instructions. May be less creative than other models and task performance can vary;

- LLaMA (Meta): open-source, customizable and computationally efficient. Limited multimodal functionality and generally lower performance on some tasks compared to proprietary models;

- Gemini (Google DeepMind): advanced reasoning and problem-solving, well-integrated with Google's ecosystem. Access may be limited, and privacy considerations are important.

Chapter 3

Problem Statement

The primary goal of this work is to explore methods for automating or supporting the conceptual modeling of data marts using the Dimensional Fact Model. Conceptual modeling is a critical step in the design of data warehouses, as it translates high-level business requirements into structured representations. This process is often time-consuming, requires extensive domain expertise and involves continuous interaction between designers and end users to resolve ambiguities and align understanding.

To address these challenges, this thesis investigates the potential of Large Language Models to assist in conceptual modeling. Specifically, the analysis focuses on prompting the LLMs with domain information in two different approaches.

Some experiments leveraging LLMs for conceptual design have already been reported in the literature [4, 11]. These studies suggest that, although LLMs can effectively support designers by rapidly generating draft solutions, substantial human involvement remains necessary to validate and refine the outcomes [21]. Furthermore, [24] describe an experiment with GPT showing that, while the integration of LLMs into human-driven conceptual design does not substantially affect output quality, it considerably shortens completion time by streamlining the process and reducing the number of required design steps.

In **supply-driven** approach [17], input consists of the logical schema of a database, representing the structure of operational sources, an example is the following:

```
CREATE TABLE SUPPLIERS (
SupplierName VARCHAR(50) PRIMARY KEY,
SupplierCity VARCHAR(50),
SupplierAddress VARCHAR(100)
);
```

```
CREATE TABLE SUPPLIES (
Brand VARCHAR(50) PRIMARY KEY,
SupplierName VARCHAR(50),
FOREIGN KEY (SupplierName) REFERENCES SUPPLIERS(SupplierName)
);

CREATE TABLE COMPUTER (
CodComputer INT PRIMARY KEY,
Brand VARCHAR(50),
Model VARCHAR(50),
FOREIGN KEY (Brand) REFERENCES SUPPLIES(Brand)
);

CREATE TABLE SOFTWARE (
CodSoftw INT PRIMARY KEY,
SoftwareDescription VARCHAR(100),
Type VARCHAR(50)
);

CREATE TABLE INSTALLATIONS (
CodComputer INT,
CodSoftw INT,
InstallationDate DATE,
PRIMARY KEY (CodComputer, CodSoftw),
FOREIGN KEY (CodComputer) REFERENCES COMPUTER(CodComputer),
FOREIGN KEY (CodSoftw) REFERENCES SOFTWARE(CodSoftw)
);
```

In **demand-driven** approach [9], input consists of a textual formalization of business requirements, describing the desired analysis from the user perspective, an example related to the previous one is given:

> Decision-makers are interested in analyzing the softwares installed on computers in a laboratory on a daily basis. Each computer has a model and a brand, each brand is supplied by a supplier who lives in a city at a given address. A software has a description and a type.

In both approaches, LLMs are tasked with generating a DFM schema, which is then **compared to a ground truth model**. The evaluation of the generated schemas is performed using standard metrics such as **precision, recall and F1-score**. In addition, custom error metrics are considered to capture domain-specific mismatches or inconsistencies that may not be fully reflected by general measures, providing a more nuanced assessment of schema quality.

The process is fully automated across different models, allowing a systematic comparison of their performance. In addition to accuracy metrics, **processing time is collected** to assess the efficiency of each model in producing complete and correct DFM schemas. Moreover, for each **input model**, execution times are recorded for both **CPU and GPU executions**. This approach allows a more precise and meaningful comparison of computational efficiency across different models and hardware configurations.

In addition to defining the general task, this work also explores the **role of prompt formulation** in both supply-driven and demand-driven approaches, given that literature also includes promising analyses on the impact of few-shot learning and prompting strategies in enhancing LLM performances [12]. Different prompt variants are tested to assess their impact on the quality of the generated Dimensional Fact Model schemas. Prompt design is inspired by guidelines from recent scientific literature, an overview in [1], and aligned with the chat-based templates of the imported models, particularly leveraging the instruction-following class to ensure consistency with the models' expected input format.

Chapter 4

System Architecture

4.1 Application Design

The system has been designed as a modular **pipeline**, starting from configuration files and culminating in the generation of evaluation metrics and visual graphs. The design is illustrated in Figure 4.1, which highlights the key modules and their interactions.

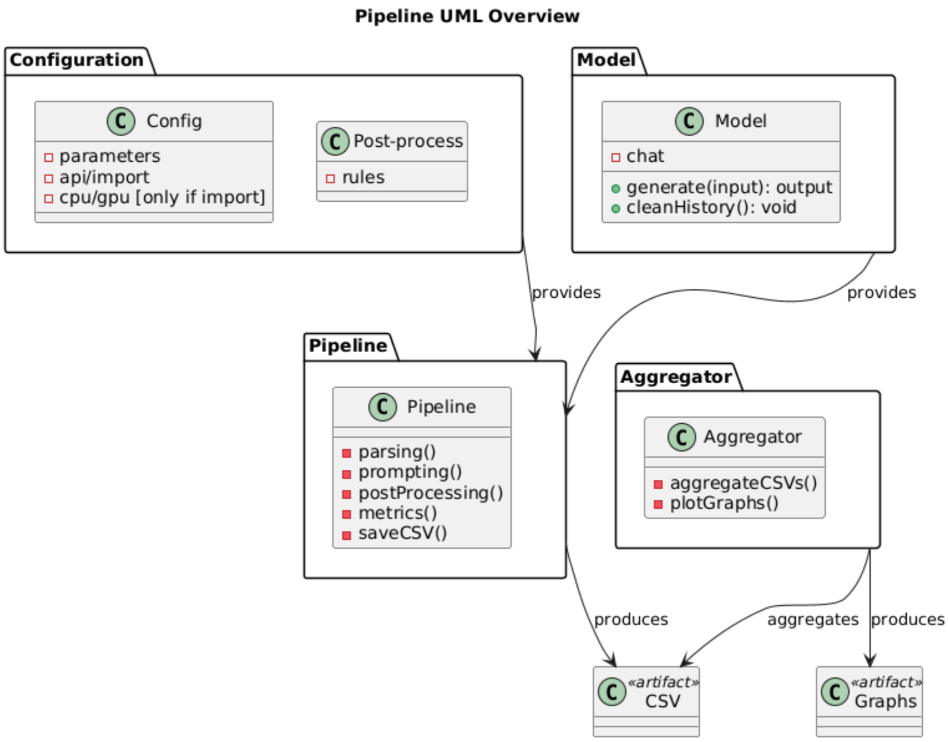


Figure 4.1: UML class diagram of the application architecture.

It is worth noting that `Pipeline` and `Aggregator` are not implemented as persistent modules in the codebase, but rather as Python scripts. The UML diagram emulates their macro steps in order to capture their role in the workflow: the `Pipeline` script executes parsing, prompting, post-processing and metrics computation, while the `Aggregator` script consolidates CSV outputs and produces visual graphs. Thus, the diagram should be interpreted as a conceptual representation of responsibilities, rather than a one-to-one mapping to software classes.

4.1.1 Pipeline Overview

The process begins with configuration files, which define the experimental setup, such as model to use, input exercise and output parameters. The pipeline consists of the following stages:

1. Configuration Parsing: system parses the configuration file, extracting parameters for model selection and prompt type;

2. Model Loading: if an imported LLM is required, the application loads it into memory, preparing either CPU or GPU execution according to the configuration, else APIs endpoints are loaded;

3. Prompting and Execution Time Collection: selected model is prompted with either supply-driven or demand-driven inputs, consisting of an example of a problem solved step-by-step and problem to solve formulation, collecting execution times for both CPU and GPU runs;

4. Output Parsing and Post-Processing: model outputs are transformed into a structured YAML schema, then refined by applying custom post-processing rules to correct common modeling errors;

5. Ground Truth Processing: reference schema undergoes the same post-processing, ensuring correct comparability with model outputs;

6. Comparison and Metrics Evaluation: models' output is compared to the ground truth, computing standard metrics and custom error categories;

7. Result Storage: evaluation results are stored as CSV files, which include metrics, custom errors, run configurations and execution times;

8. Aggregation and Visualization: separate script aggregates all CSV outputs and generates graphs to highlight differences among models and configurations.

The main workflow of the application is illustrated in Figure 4.2, which shows the flow of data through the system.

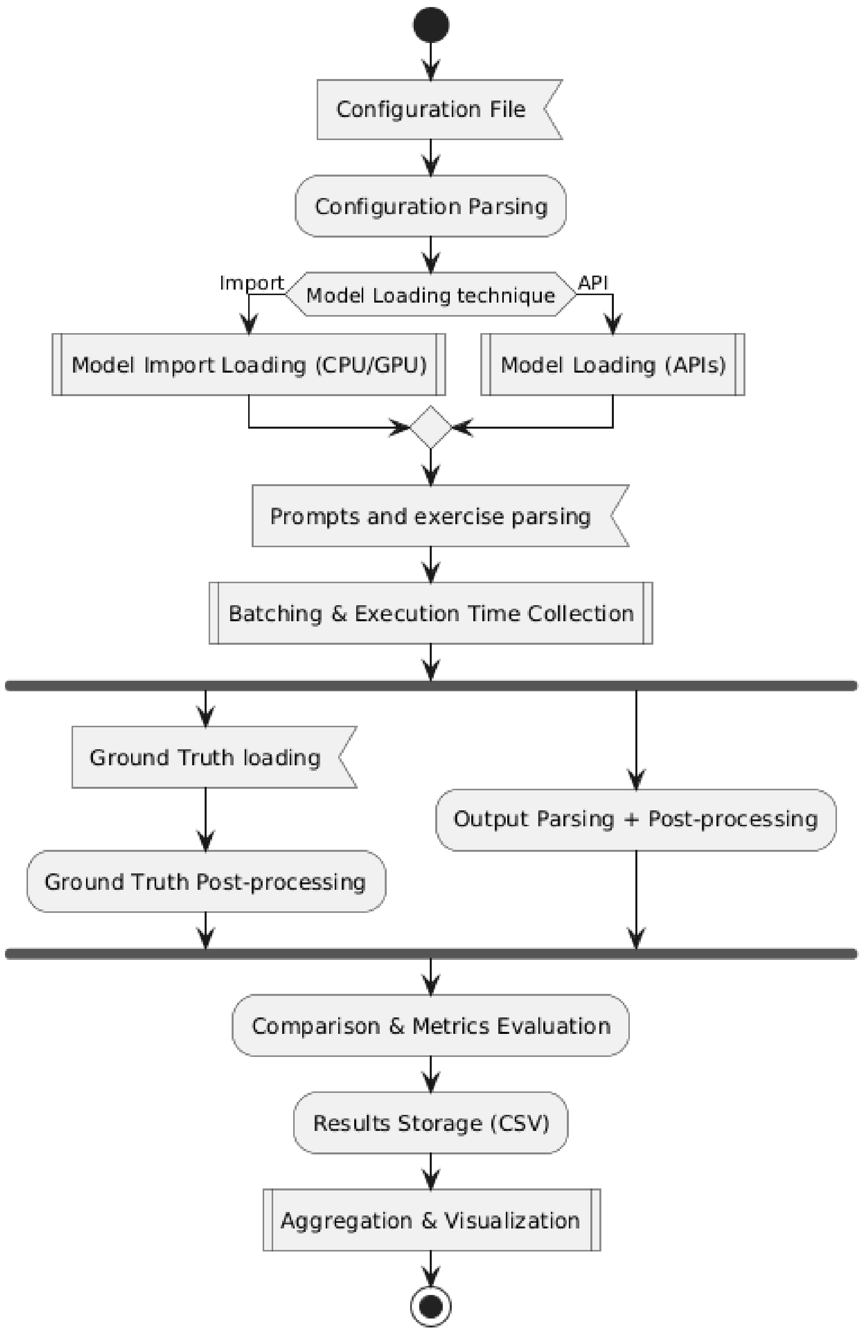


Figure 4.2: UML activity diagram of the application architecture.

4.1.2 Model Abstraction

To ensure a flexible and unified handling of both imported and API-based Large Language Models, an object-oriented abstraction was implemented in *Python*. The central component is a `Model` class, which provides a consistent interface for generation tasks regardless of the underlying source of the model.

The `Model` object is responsible for encapsulating the following elements:

- **Model and Tokenizer**: for imported models only, the Hugging Face `transformers` library is used to load both the model and its tokenizer from the local environment or directly from the Hugging Face Hub;
- **Generation Function**: a uniform `generate()` method is defined to perform inference. This function transparently supports both imported models, on CPU or GPU, and API-based models, invoked through REST endpoints;
- **Chat History**: the object maintains the conversational history to support multi-turn prompting and consistency across requests;
- **Configuration Handling**: model instances are initialized from configuration files that specify the model type (import or API), its parameters and any runtime options.

This design enables the pipeline to treat all models uniformly, while internally abstracting the differences between imported and API-based usage. Whereas an API-based model is wrapped by the same `Model` object, but internally relies on HTTP requests to the APIs. From the perspective of the pipeline, both are invoked through the same `generate()` interface, making the execution fully interchangeable.

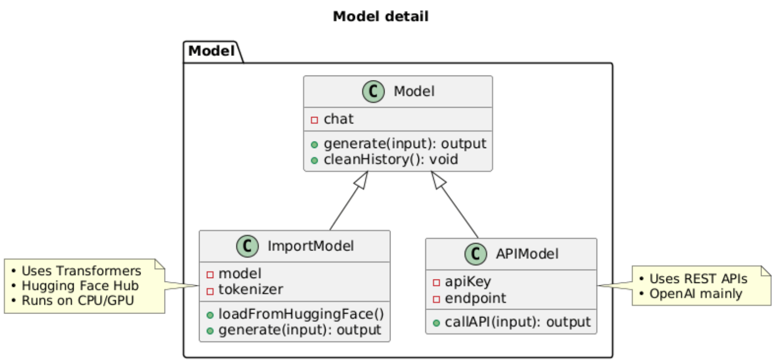


Figure 4.3: UML diagram of the `Model` class.

4.1.3 Input

The entry point of the system is the prompt, which defines how models are queried under both supply-driven and demand-driven approaches. A model-specific variant (e.g. the `GPT` block) replicates the same logical content but may reorder or adapt wording to match a particular model's recommendation style (role ordering or phrasing). Internally, the pipeline can load either the generic 'base' few-shot prompt or the model-specific prompt defined in the configuration.

Role-based prompting

Prompts in the pipeline are consistently expressed as lists of $key->values$ structures, each with a `role` and a `content`. This design aligns with the role-based chat paradigm supported by modern LLMs. The main roles are:

- **system**: establishes the context and global constraints under which the model must operate. Typical use: define the persona, enforce the required output format and restrict the style;
- **user**: expresses the actual task to be solved. This includes both the high-level problem specification and detailed constraints. In few-shot settings, user messages may also introduce worked examples of tasks;
- **assistant**: provides illustrative completions in the case of few-shot prompting. These messages act as demonstrations of the expected reasoning steps or final outputs. In zero-shot settings, this role is usually omitted, and the model is expected to generate the output directly after the last user message.

Ordering constraints

The canonical sequence is:

1. one or more `system` messages;
2. one `user` message stating the task;
3. optionally, alternating pairs of `assistant` and `user` messages for few-shot demonstrations;
4. a final `user` message, after which the model is expected to generate its completion.

While most frameworks allow some flexibility, deviating from this order can lead to unstable or incoherent outputs. The pipeline therefore enforces this structure to ensure reproducibility across experiments.

Standardized structure

Each prompt follows a standardized structure:

- an **introductory block**, common to all prompts, which specifies the expected output format in `YAML` and provides high-level instructions to the model, detailed in 4.1;
- a **task-specific block**, which encodes the particular research question or design objective.

```yaml
- role: system
  content: |
    You are a data warehouse designer.
- role: user
  content: |
    Carry out supply-driven conceptual design of a multidimensional cube using
    ↪ the DFM (Dimensional Fact Model), starting from the following source
    ↪ schema (for supply-driven) / requirements (for demand-driven).
```

Listing 4.1: Introductory block for all prompts.

Supply-driven prompts

The goal of this prompt is to carry out a supply-driven conceptual design task by deriving the fact and candidate measures of a multidimensional cube starting from a source relational schema. Three variants have been defined, each aligned with a specific research question:

- **RQ2**: this prompt is formulated as a **zero-shot instruction**, with no explicit examples included. The decision to explore a zero-shot setting is to test the model's ability to generalize from detailed task descriptions alone. In particular, (i) the expected output is a highly structured `YAML` format, so providing few-shot examples could bias the generation towards incorrect or overly specific schemas; and (ii) the prompt already specifies in detail the structure and constraints of the output, effectively encoding much of the guidance that examples would otherwise provide;
- **RQ3-dec**: this prompt is **few-shot**, including one or more example schemas together with their intermediate FD lists and the resulting DFM YAML. This is motivated by two practical reasons:

  - **Procedural complexity**: the model must apply a deterministic sequence of graph operations, such as FD extraction, transitive collapse and root selection, that is hard to convey only by natural-language constraints. Worked examples expose the intended algorithmic behavior and the expected output shape;
  - **Format & algorithm coupling**: Examples demonstrate not only the output format but the transformation steps that produce that format; they therefore reduce ambiguity about how to treat composite keys, foreign keys and composite dependencies.

  To mitigate the drawback about examples that may biasing the model toward specific patterns, examples are chosen to cover different structural cases; moreover, the prompt point out the expected YAML structure. A summarization is provided in 4.2;

- **RQ3-alg**: this prompt is **few-shot** too, with the objective to infer dependencies and construct the DFM hierarchy by explicitly following an **algorithmic procedure**. This variant prescribes a step-by-step graph construction method that the model must apply to a given relational schema to produce the final DFM YAML.

  Key points about RQ3-alg:

  - **Algorithmic specification**: the prompt defines a deterministic sequence of operations the model must perform: identify the fact candidate, initialize the root node, expand nodes via foreign-key links, optionally add domain-informed FDs, rename the root to the fact name and select numerical attributes as measures. This explicit specification reduces ambiguity about how to navigate foreign-key chains and how to treat composite attributes;
  - **Worked examples included**: the prompt contains full worked examples, from input schema to final YAML, to demonstrate both the transformation steps and the exact output shape. Examples are necessary here because the task requires the model to emulate a multi-step algorithmic process rather than produce a single mapping from input to output.

  To mitigate the drawback about examples that may biasing the model toward specific patterns, examples are chosen to cover different structural cases; moreover, the prompt point out the expected YAML structure. A summarization is provided in 4.3.

```yaml
- role: user
  content: |
    ...
    Data-driven conceptual design of a multidimensional cube starting from a source relational schema S should be carried out via the following steps.
    1. Find all the functional dependencies ...
    2. Foreign keys substitution ...
    3. Find the roots picking the one that allows reaching the greatest number of attributes by navigation FDs in the
    correct direction, and draw the corresponding tree ...
    The output I expect is a DFM schema in YAML formatted as follows:
    (1) "fact" including a "name", listing the name of the root;
    (2) all measures (i.e., all numerical attributes directly connected to the root) are listed inside a "measures" tag, each as empty item containing a "name" tag;
    (3) all functional dependencies between attributes in a hierarchy are listed inside a "dependencies" tag:
    each as an item containing a "from" tag, listing finer attribute(s), a "to" tag, listing coarser attribute, and optionally a "role" tag;
    (4) the "dependencies" list also includes an item from the fact to each measure;
    (5) if a dependency is from and/or to a combination of attributes, they should be comma-separated;
    (6) all attributes and measure names must be prefixed by the name of the table they belong to.
- role: assistant
  content: |
    Let this relational schema be given:

    CREATE TABLE ...

    After step 1, the list of FDs is the following:
    - PURCHASE.storeId,PURCHASE.artId,PURCHASE.date->PURCHASE.storeId
    ...

    After step 2, the list of FDs is changed as follows:
    - PURCHASE.storeId,PURCHASE.artId,PURCHASE.date->PURCHASE.storeId and PURCHASE.storeId->STORES.storeId
    are replaced with PURCHASE.storeId,PURCHASE.artId,PURCHASE.date->STORES.storeId
    - ...

    After step 3, the YAML expression of the DFM schema is obtained as follows:
    fact:
    name: PURCHASE
    measures:
    - name: PURCHASE.quantity
    - ...
    dependencies:
    - from: TYPES.typeId
        to: TYPES.typeName
    - ...
- role: user
  content: |
    Here is another example; ...
```

Listing 4.2: RQ3-dec summarization.

```
- role: user
  content: |
    ...
    The first step is to create a directed
    acyclic graph G where each node corresponds to a (simple or composite)
    ↪  attribute of D and each arc
    corresponds to a functional dependency (FD). You can do this via the
    ↪  following steps:

    0. Find a fact R ...
    1. INITIALIZE: Add to G the primary key of R, R.K, as a node. ...
    2. EXPAND R: Add to G an arc from R.K to each other attribute of R that is
    ↪  not a foreign key. ...
    3. REARRANGE G: In some cases, some additional FDs may be inferred ...
    4. RENAME THE FACT: ...
    5. CHOOSE MEASURES: ...

    The output I expect is a DFM schema in YAML formatted as follows ...

    Let this relational schema be given:
    CREATE TABLE ...
    0. The fact here is ...
    1. INITIALIZE: G is initialized with node ...
    2. EXPAND PURCHASE: ...
    2.1 EXPAND STORES: ...
    2.1.1 EXPAND REGIONS: ...
    2.2 EXPAND ARTICLES: ...
    2.2.1 EXPAND TYPES: ...
    3. REARRANGE G: ...
    4. RENAME THE FACT: ...
    5. CHOOSE MEASURES: ...

    The YAML expression of the DFM schema obtained in the end is the following:

    fact:
    name: PURCHASE
    measures:
    - name: PURCHASE.quantity
    - ...
    dependencies:
    - from: TYPES.typeId
        to: TYPES.typeName
    - ...

    Here is another example ...
```

Listing 4.3: RQ3-alg summarization.

Demand-driven prompts

The goal of these prompts is to support a demand-driven conceptual design task, where the construction of the multidimensional schema starts from a set of analytical requirements. The main challenge is the higher degree of ambiguity in requirements compared to schema-driven design: requirements may leave implicit which entities are facts, which quantities should be treated as measures, and how attributes should be organized in hierarchies Two variants have been defined, each aligned with a specific research question:

- **RQ4**: this prompt is formulated as a **zero-shot instruction**. It specifies a strict YAML format and enforces the use of role-based dependencies to ensure consistent hierarchy construction. The model is constrained by a clear specification of output format, which acts as a structural guide.

  The resulting YAML captures:

  - the `fact` identified from the analytical goal,
  - a list of `measures` inferred from the quantitative aspects of the requirements,
  - the `dependencies` encoding many-to-one associations among attributes as well as the connections from the fact to each measure and dimension.

  The whole prompt is stated in 4.4;

- **RQ5**: this prompt is **few-shot**, and emphasizes semantic disambiguation and the explicit encoding of modeling choices that are commonly required in real-world requirements.

  Key aspects of RQ5:

  - **Refinement-focused task**: the prompt instructs the model how to treat repeated concepts by merging them into a single node and annotating incoming arcs with `role` labels, how to treat identifiers, creating only the identifier node rather than separating entity node, and how to represent many-to-one relationships and shared hierarchies;
  - **Worked examples included**: provides multiple demand-driven examples from requirements to final YAML to enhance disambiguation and supply modelling conventions that are subtle and context-dependent. Examples highlight how to: assign roles to duplicate concepts, choose which attributes are measures, create shared sub-hierarchies and express composite or role-bearing dependencies.

  A summarization is provided in 4.5.

```yaml
- role: user
  content: |
    Carry out demand-driven conceptual
    design of a multidimensional cube using the DFM (Dimensional Fact Model),
    ↪ starting from the following
    requirements.
    The output I expect is a DFM schema in YAML formatted as follows:
    (1) the fact is a "fact" tag including a "name" tag;
    (2) all measures are listed inside a "measures" tag, each is an empty item
    ↪ containing a "name" tag;
    (3) all many-to-one associations between attributes in a hierarchy are
    ↪ listed inside a "dependencies"
    tag: each is an empty item containing a "from" tag, listing the finer
    ↪ attribute, a "to" tag, listing
    the coarser attribute, and optionally a "role" tag;
    (4) the "dependencies" list also includes an item from the fact to each
    ↪ dimension, and one from
    the fact to each measure.

    Return only the YAML without any further information/explanation.
```

Listing 4.4: RQ4 summarization.

```yaml
- role: user
  content: |
    Your task is to create the conceptual schema for a multidimensional cube in a
    demand-driven fashion, i.e., starting from the end-user requirements, using the DFM.
    To this end, keep in mind that a DFM is a connected graph where the fact is a
    node in which no arcs enter. The fact is ...
    The arcs within a hierarchy are ..
    When you have the same concept used twice ...
    When the requirements specify that a concept is identified by an attribute ...

    The output I expect is a DFM schema in YAML formatted as follows:
    (1) the fact is a "fact" tag ...
    (2) all measures are ...
    (3) all many-to-one relationships ...
    (4) the "dependencies" list ...

    Let these requirements be given:
    Decision-makers are interested in analyzing, in terms of quantity and cost ...
    A warehouse has a unique name, an address ... Articles are identified by a code ...

    Here, SHIPMENT is the fact, ...
    Article code, warehouse, and day are dimensions ...

    The YAML expression of the DFM schema obtained is the following:

    fact:
    name: SHIPMENT
    measures:
    - name: quantity
    - ...
    dependencies:
    - from: articleCode
        to: articleName
    - ...

    Here is a second example; requirements ...

    Please apply this process to the following source schema, and create the YAML representation of the resulting DFM schema.
    You must only output the YAML file ...
```

Listing 4.5: RQ5 summarization.

4.1.4 Output Structure

The output generated by the system is always expressed in YAML format, a full demand-driven example is supplyed below, exercise text in 4.6, prompt in 4.7, output obtained from GPT-4 in Figure 4.4a and ground truth for comparison in Figure 4.4b.

```yaml
text: |
  Decision-makers are interested in analyzing the softwares installed on computers in a laboratory
  on a daily basis. Each computer has a model and a brand, each brand is supplied by a supplier who
  lives in a city at a given address. A software has a description and a type.
```

Listing 4.6: Exercise example.

```yaml
- role: system
content: |
    You are a data warehouse designer that outputs YAML-formatted data. Please generate a YAML response for the following input.
- role: user
content: |
    Carry out demand-driven conceptual
    design of a multidimensional cube using the DFM (Dimensional Fact Model), starting from the following
    requirements.
    The output I expect is a DFM schema in YAML formatted as follows:
    (1) the fact is a "fact" tag including a "name" tag;
    (2) all measures are listed inside a "measures" tag, each is an empty item containing a "name" tag;
    (3) all many-to-one associations between attributes in a hierarchy are listed inside a "dependencies"
    tag: each is an empty item containing a "from" tag, listing the finer attribute, a "to" tag, listing
    the coarser attribute, and optionally a "role" tag;
    (4) the "dependencies" list also includes an item from the fact to each dimension, and one from
    the fact to each measure.

    Return only the YAML without any further information/explanation.
```

Listing 4.7: Prompt example.

```yaml
fact:
  name: SoftwareInstallation
measures:
- name: installationCount
dependencies:
- from: SoftwareInstallation
  to: Computer
- from: SoftwareInstallation
  to: Software
- from: SoftwareInstallation
  to: installationCount
- from: Computer
  to: Model
- from: Model
  to: Brand
- from: Brand
  to: Supplier
- from: Supplier
  to: City
- from: City
  to: Address
- from: Software
  to: Description
- from: Description
  to: Type
```

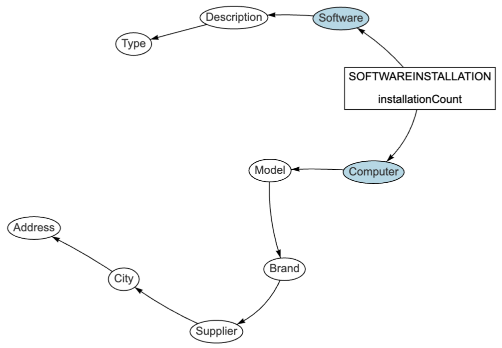
(a) Output example.

```yaml
fact:
  name: INSTALLATION
measures:
dependencies:
- from: INSTALLATION
  to: Date
- from: INSTALLATION
  to: Computer
- from: INSTALLATION
  to: Software
- from: Computer
  to: Model
- from: Computer
  to: Brand
- from: Brand
  to: Supplier
- from: Supplier
  to: City
- from: Supplier
  to: Address
- from: Software
  to: Description
- from: Software
  to: Type
```

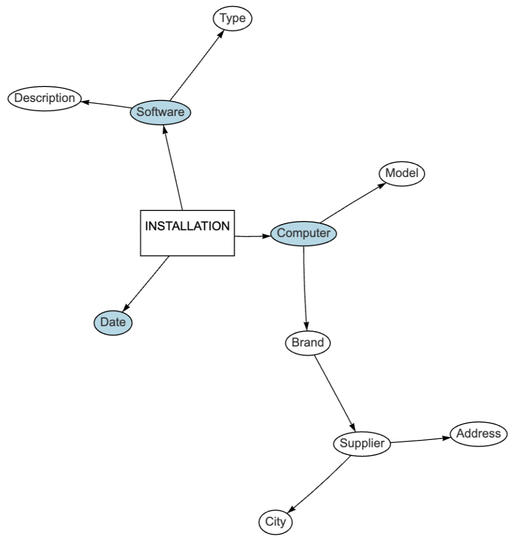
(b) Ground truth of the example.

Figure 4.4: Comparison of output & ground truth.

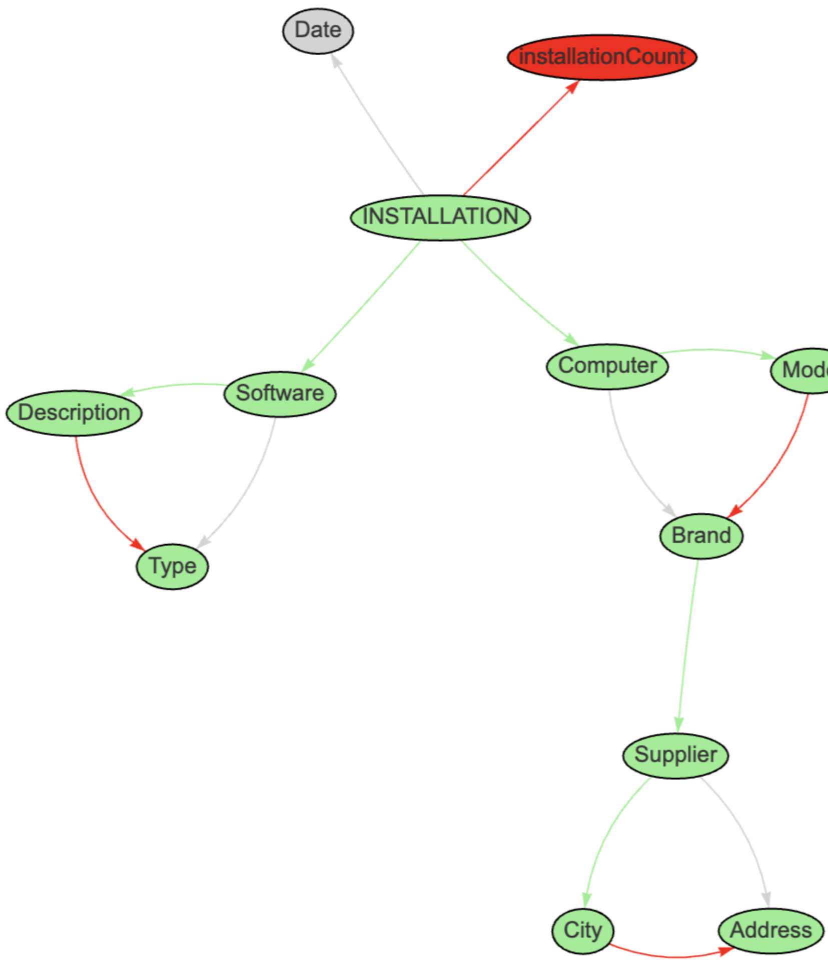
In this structure, the `fact` represents the central phenomenon under analysis, while `measures` list the quantitative values to be observed. The `dependencies` block encodes the relationships between attributes. Conceptually, every `name` can be interpreted as a *node* of a directed graph, and each `from` $\rightarrow$ `to` pair defines an *edge*. This graph-based interpretation highlights how facts, measures and dimensions are interconnected, ensuring that the resulting schema can be easily visualized as a network of relationships, as highlighted in Figure 4.5.



(a) Output graph.



(b) Ground truth graph.



(c) Comparison graph, highlighting green nodes and edges for TP, grey for FN and red for FP.

Figure 4.5: Graphs for representation.

4.1.5 Post-Processing

Both model-generated outputs and ground truth schemas undergo a post-processing phase before comparison. This step ensures that variations in terminology, naming conventions, or minor structural differences do not bias the evaluation metrics.

The post-processing operations are based on two types of rules:

- **Equivalence rules**, which map different node labels to a single canonical representation;
- **Ignore rules**, which identify attributes or nodes that should be excluded from the evaluation, either because they are non-essential for the exercise or because they represent redundant details.

Rules are structured in two layers:

- **Common rules**, applied across all exercises;
- **Exercise-specific rules**, tailored for each individual task, to account for domain-specific variations and vocabulary.

This strategy guarantees that the evaluation focuses on the conceptual correctness of the schema rather than superficial naming differences, thus enabling a fairer and more accurate measurement of model performance.

It's worth mentioning that these rules differ for demand and supply approaches, as shown in 4.8.

```yaml
common:
  demand:
    equals:
    - Date:
      - day
    ignore:
      - count
      - month
      - year
  supply:
    equals: []
    ignore: []

excercise-number:
  demand:
    equals:
    - INSTALLATION:
      - INSTALLATIONS
      - SoftwareInstallations
      - SoftwareInstallation
    - model:
      - computerModel
    - brand:
      - computerBrand
    - supplier:
      - brandSupplier
    - city:
      - supplierCity
    - address:
      - supplierAddress
    - description:
      - softwareDescription
    - type:
      - softwareType
    ignore: []
  supply:
    equals: []
    ignore: []
```


Listing 4.8: Exercise specific and common rules example.

Each rule follows a simple interpretation logic:

- **equals**: specifies sets of node names that must be considered equivalent. For instance, in the example above, `INSTALLATION`, `INSTALLATIONS` and `SoftwareInstallation(s)` all map to the same conceptual node that is `INSTALLATION`. During comparison, these variations are treated as a single element;
- **ignore**: lists node names that must be excluded from the evaluation. These are often auxiliary or redundant attributes (e.g. `count`, `month`, `year`) which would otherwise affect accuracy metrics without adding meaningful information;
- **common rules**: applied to all exercises under the same approach (supply-driven or demand-driven). They ensure consistency across tasks and reflect domain-specific simplifications;
- **exercise-specific rules**: tailored to a single exercise and override or extend the common rules. This makes it possible to normalize variations that are unique to a given dataset or requirement specification.

By interpreting the rules in this way, both the model output and the ground truth are aligned onto a shared semantic representation, ensuring that metrics are not biased by irrelevant discrepancies.

4.2 Aggregation and Visualization

After the computation of metrics for each run, results are collected into `CSV` files, one for each model–approach–prompt and a given label (if not given, a timestamp is provided) configuration. To provide a comprehensive overview, these results are then aggregated and analyzed through a set of six types of graphs, each of which highlights a different perspective on model performance.

- **Boxplot of F1-score for edges by exercise number.** This visualization illustrates the distribution of F1-scores for edge prediction across all runs of a given exercise. It highlights median performance as well as the variability, enabling a quick comparison between exercises.
  An example in Figure 4.6a;
- **Boxplot of F1-score for nodes by exercise number.** Similar to the edge-based boxplot, but focusing on node prediction. This helps in identifying whether node modeling is generally more stable or prone to errors

compared to edge modeling.
An example in Figure 4.6b;

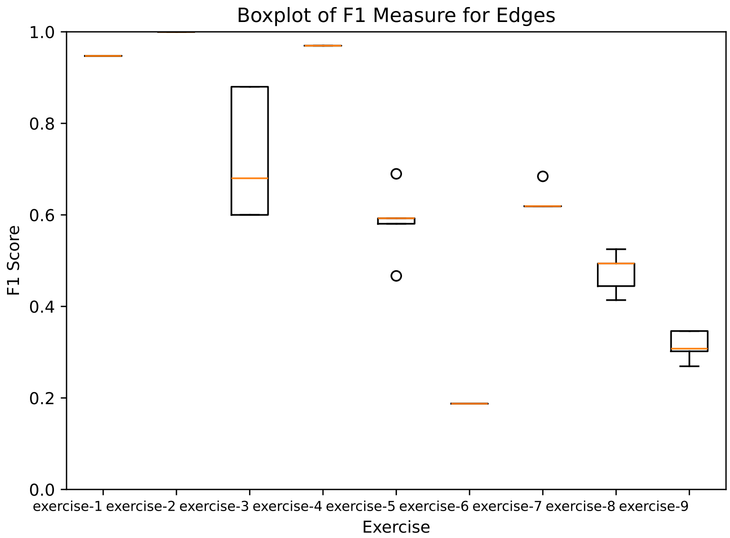
- **Line graph of average F1-score for nodes and edges by exercise number.** This aggregated graph displays how average F1 evolves across exercises.
An example in Figure 4.6c;

- **Bar graph of average precision and recall for nodes by exercise number.** Precision and recall offer complementary insights: high precision indicates few false positives, while high recall shows coverage of relevant nodes. This graph makes it possible to see trade-offs between the two.
An example in Figure 4.7a;

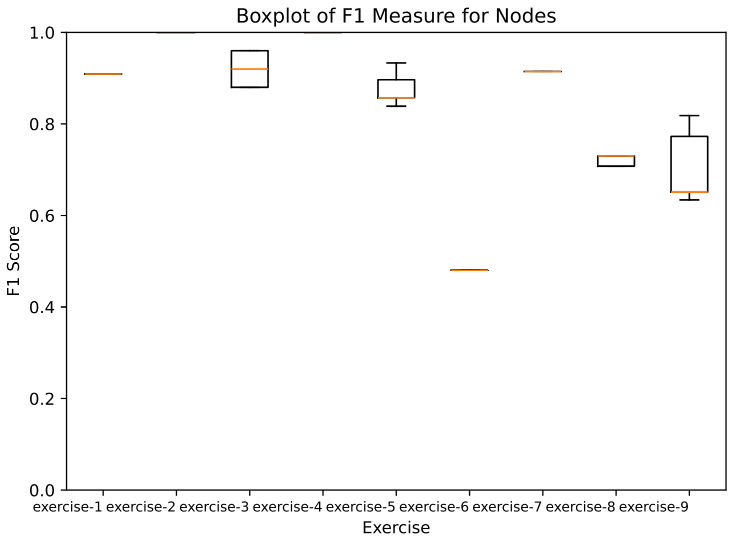
- **Bar graph of average precision and recall for edges by exercise number.** Like the node-focused bar graph, but for edge relationships. Differences here often reveal whether LLMs capture structural relationships consistently or only partially.
An example in Figure 4.7b;

- **Scatter plot of average F1 (nodes and edges) vs. average execution time.** This graph captures the trade-off between predictive performance and computational efficiency.
An example in Figure 4.7c.

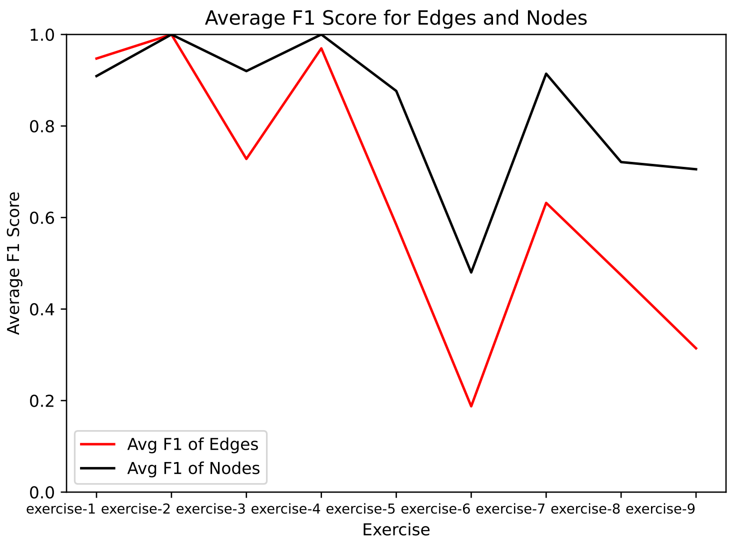
Together, these graphs provide both a fine-grained and aggregated perspective, enabling a robust comparison between supply-driven and demand-driven approaches, across prompts and models.



(a) Boxplot of F1-score for edges.

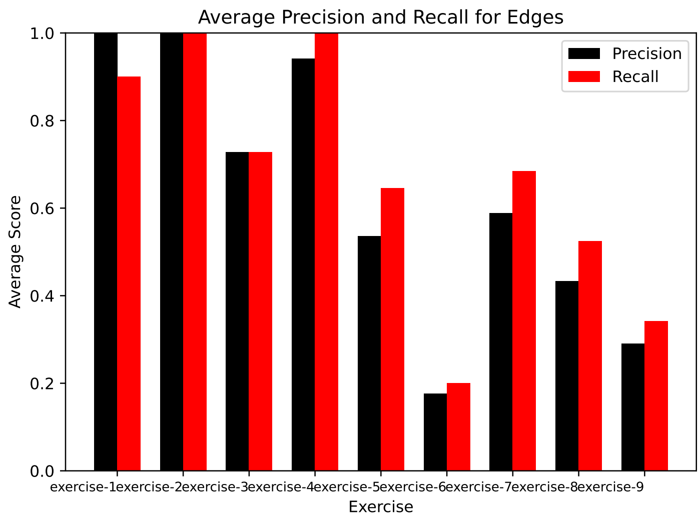


(b) Boxplot of F1-score for nodes.

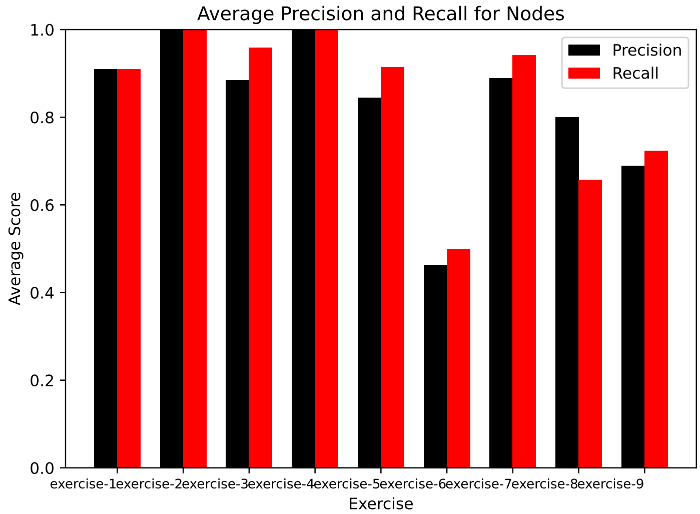


(c) Graph F1-score aggregating nodes and edges.

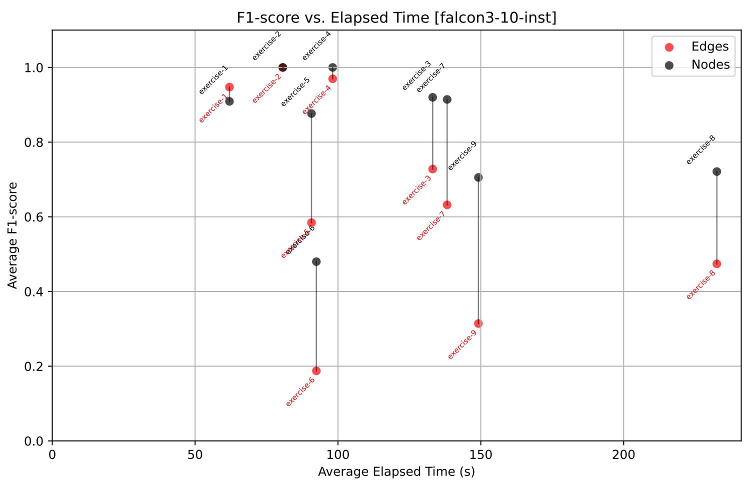
Figure 4.6: Graphs produced by Falcon model for aggregation.



(a) Graph Precision and Recall for edges.



(b) Graph Precision and Recall for nodes.



(c) Graph F1-score and time.

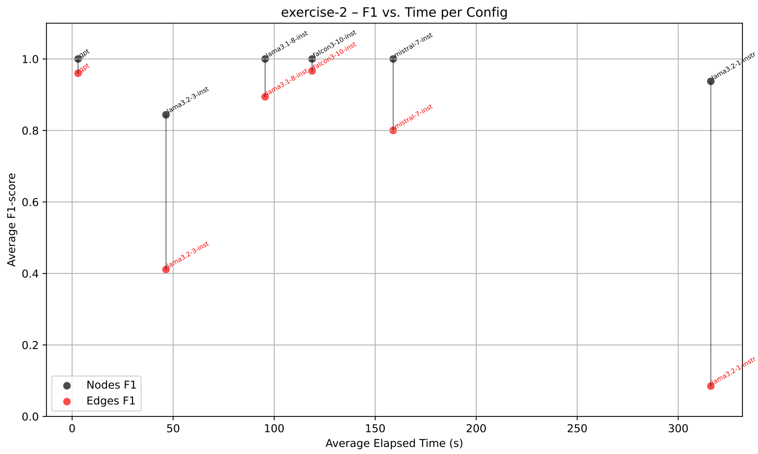
Figure 4.7: Graphs produced by Falcon model for aggregation.

Beyond the core set of six graphs previously described, dedicated scripts were developed to produce additional visualizations that provide further insights into model behavior and performance trade-offs.

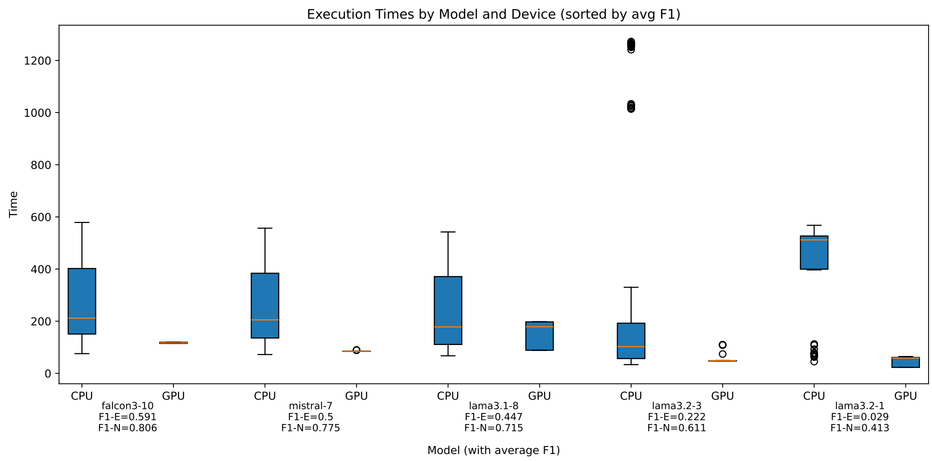
- **Global F1 vs. Execution Time.** A scatter plot that merges all results of a single exercise by considering the average F1-score for nodes and edges on the y-axis and the average execution time on the x-axis. This visualization highlights the balance between predictive accuracy and computational efficiency across models.
  An example in Figure 4.8a;

- **Execution Time Boxplot by Model.** A boxplot in which execution times are reported on the y-axis, while the x-axis lists all imported models sorted according to their overall average F1-score, aggregated over nodes and edges. Results are further split between CPU and GPU execution, allowing for a clear comparison of hardware impact on performance.
  An example in Figure 4.8b;

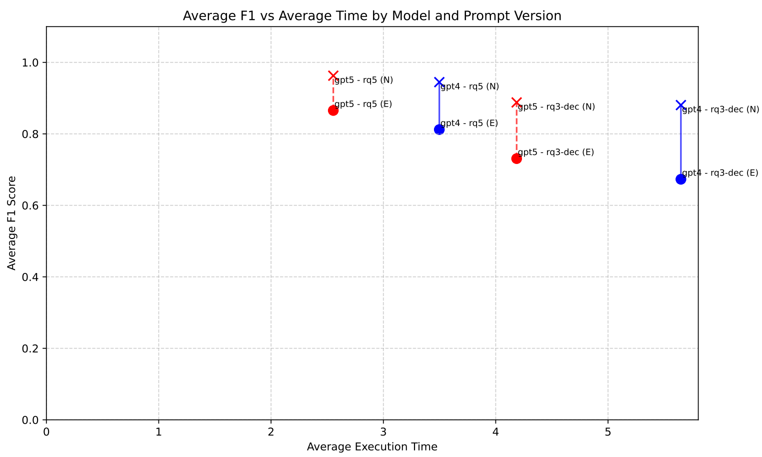
- **GPT-4 vs. GPT-5 Comparison.** A comparative plot dedicated to GPT-4 and GPT-5, reporting the average F1-score on the y-axis and execution time on the x-axis. The graph is split by different prompts, thereby reflecting the variations across supply-driven and demand-driven approaches. This enables a direct assessment of the impact of prompting strategies on performance and efficiency.
  An example in Figure 4.8c.



(a) Exercise specific graph aggregating average f1 and execution times for different models.



(b) Aggregation over all exercises for execution times and f1 for CPU and GPU execution.



(c) Comparison for GPT versions in terms of f1 and response times.

Figure 4.8: Additional graphs aggregating execution times and accuracy metrics.

These additional visualizations complement the primary set of graphs by offering both a broader overview of the trade-off between accuracy and execution time and a focused comparison on the most advanced models.

Chapter 5

Implementation

This chapter presents the implementation of the system outlined in the previous sections, translating the conceptual pipeline into a fully functional prototype. Particular attention is devoted to design principles such as **modularity**, **reusability** and the **single responsibility principle**, which collectively ensure a maintainable and extensible architecture. The configurability of the system is enabled by the principle of separating configuration from code: this allows parameters such as API keys, configurations and runtime options to be modified without altering the source code, improving **maintainability**, **portability** and **security**.

5.1 Environment and Tools

The system was developed using modern programming languages and widely adopted libraries, enhancing flexibility and reproducibility.

- **Programming Languages**: Python 3.12, JavaScript, Bash;
- **Core Libraries**: Hugging Face `transformers` and PyTorch for model handling and inference; `requests` for API-based interaction; `yaml` for configuration parsing; `pandas` and `matplotlib` for data aggregation, analysis and visualization;
- **Data Formats**: configuration, exercises, prompts and outputs stored in `YAML`, evaluation metrics and aggregated results maintained in `CSV`, graph visualization through `PDF`;
- **Dependency Management**: `Poetry` for package management, dependency resolution and to publish to **PyPi**.

5.1.1 Supporting Technologies

To ensure reliability, reproducibility and ease of deployment, the following supporting technologies were integrated:

- `GitHub` for version control and **Continuous Integration**;
- `Docker` for virtualization and environment encapsulation;
- `Portainer` as a container management interface to monitor and manage Docker deployments;
- `Windows Remote Desktop` for secure access to on-premise university laboratory machines;
- `Trello` to organize and track work during the project, creating boards and cards for tasks and monitoring progress;
- `HuggingFace` as a model hub and repository for importing pre-trained LLMs;
- `Node.js` as a runtime environment to support release automation workflows.

5.2 Pipeline Implementation

The pipeline follows the architecture described in Figure 4.1. It is implemented as a Python script that executes the following steps.

5.2.1 Configuration Parsing

The entry point of the pipeline is the configuration parser, responsible for loading and validating the parameters required for execution. Configuration files are written in `YAML` format, as this ensures human readability while supporting nested structures such as rules, model parameters and experiment metadata.

Parsing is implemented through the `PyYAML` library. The parser loads the specified file, checks for required fields and applies default values where necessary. Configuration objects are then passed forward to subsequent stages of the pipeline.

A complete example of the configuration file is reported in 5.1, illustrating all the parameters required for the pipeline, including model specifications, prompting options, rules for post-processing and output settings.

```yaml
use: 'api' # import or api
debug_prints: false

model_import:
  name: 'falcon'
  temperature: 0.2
  max_new_tokens: 4000
  do_sample: true
  top_p: 0.9
  device: 'cpu'

model_api:
  name: 'gpt'
  label: 'gpt4o' # [gpt3-5, gpt4o]
  deployment: 'gpt-4o' # Deployment name for azure distribution [test-gpt-35, test-gpt-4o]
  api_version: '2024-04-01-preview' # Used with GPT only up to now [2024-02-15-preview, 2024-04-01-preview]
  max_tokens:
  n_responses: 1
  temperature: 0.1
  stop:
  top_p: 0.9
  top_k: 5

output:
  dir_label: 'test'

exercise:
  name: ['exercise-1']
  version: 'sql'
  prompt_version: 'rq3-alg-base'
  number: ['1'] # If not stated, obtained as last digit in ex. name config
```

Listing 5.1: Configurations for pipeline.

Configuration Parsing through Command Line Arguments

In addition to YAML files, the pipeline supports configuration overrides via **command line arguments**. This feature makes it possible to automate large-scale experimental runs without manually editing configuration files. Command line options are parsed using Python's `argparse` library and, when provided, they take precedence over the corresponding values defined in the YAML configuration. This mechanism ensures both reproducibility, with explicit configuration files, and flexibility by enabling batch executions with parameter variations.

An example of execution overriding model and prompt directly from the command line is shown in 5.1.

Listing 5.1: Example of pipeline execution overriding YAML parameters.

```
python pipeline.py \
    --n_runs 1 \
    --exercises '1 2 3 4 5 6 7 8 9' \
    --p_version rq3-dec \
    --exercise_version sql \
    --model llama-3.1-8B-inst-hf \
    --model_loading import \
    --model_label lama3.1-8 \
    --dir_label test \
    --device gpu
```

Sensitive credentials concern

Several configurations involve sensitive credentials, such as API keys for remote model access. To address this securely, the following strategy was adopted:

- A template file (e.g. `config-example.yaml`) is provided in the repository, documenting the expected structure of configuration files;
- The actual expected configuration file (e.g. `config.yaml`) is excluded from version control through `.gitignore`, preventing accidental exposure of credentials;
- Users cloning the repository are instructed to copy the template and fill in their own credentials locally.

This approach ensures both security and reproducibility: the pipeline runs with the required credentials while avoiding any leakage of sensitive information in public repositories.

5.2.2 Model Loading

The entry point for model usage within the pipeline is the `Model` class, detailed in Listing 5.2, which encapsulates both initialization and generation, providing a uniform interface to heterogeneous LLMs. This abstraction ensures that, regardless of whether a model is imported locally through Hugging Face or accessed remotely via APIs, the pipeline interacts with it in the same way.

Listing 5.2: Excerpt of the unified Model class handling both imported and API-based models.

```python
class Model:
    # Initialize chat, load model and generation function, device is for cpu or gpu
    def __init__(self, name, config, key, device, debug_print):
        self.chat = []
        if use_import:
            self.model, self.tokenizer = load_model_and_tokenizer(name, key, device)
            self.generate = load_generate_import_function(name, self.model, self.tokenizer, config, debug_print, device)
        elif use_api:
            self.model = load_model_api(name, key)
            self.generate = load_generate_api_function(name, self.model, config, debug_print)
    def batch(self, prompt):
        try:
            m_output = self.generate(self.chat)
        except Exception as e:
            print(f'Model batch error [{e}]')
        # Add prompt in chat
        self.chat.append(get_chat_entry('assistant', m_output, self.name))
        return m_output
    # Refresh chat
    def refresh_session(self):
        self.chat = []
```

Initialization and Loading

At instantiation, the `Model` object interprets the configuration and selects the appropriate loading strategy, supplying credentials if required as exposed in 5.2, but it's worth mentioning that all HuggingFace's models are grouped under `hf` key as `my-new-model-name` for simplicity.

- **Imported models**: The class loads both model and tokenizer through Hugging Face's `transformers` or a preferred hub as in Listing 5.3;
- **API models**: The class establishes a connection to the specified endpoint, reading authentication and endpoint details from the YAML configuration as in Listing 5.4.

```yaml
my-new-model-name:
  key:
    api: my-api-key
    import: my-import-key
```

Listing 5.2: Credentials configuration for new models.

Listing 5.3: Load import models.

```
def load_model_and_tokenizer(model_name, key, device):

    match model_name:
        # Add model name here and precise model name in
           hub
        case 'my-model':
            m_name = 'my-model-name'
        case _:
            raise Exception('Model not found')

    model_already_downloaded,
       tokenizer_already_downloaded = ...

    # Torch type optimized for CPU or GPU usage
    torch_type_to_use = torch.float16 if torch.cuda.
       is_available() and device == 'gpu' else torch.
       float32

    if model_already_downloaded:
        # Load the model and tokenizer from the saved
           directory
        model, tokenizer = load(model_directory,
           torch_dtype=torch_type_to_use)
    else:
        # Download and load the model and tokenizer from
            Hub
        model, tokenizer = load(m_name, torch_dtype=
           torch_type_to_use, token=key)
        # Save the model and tokenizer
        model.save_pretrained(model_directory);
           tokenizer.save_pretrained(model_directory);
    # Move to GPU only if available and device is set
       for GPU
    if cuda_is_available() and device == 'gpu':
        model.to('cuda')

    return model, tokenizer
```

Listing 5.4: Load api models.

```
def load_model_api(name, key):
    match name:
        case 'gpt':
            openai.api_key = key
            return openai
        case 'gemini':
            genai.configure(api_key=key)
            return genai.GenerativeModel('gemini-1.5-
               flash')
        case _:
            raise Exception('Model not found')
```

This modular design makes the system extensible: to integrate a new model, it is sufficient to specify its name and provide the import function for model and tokenizer or API client.

Batch Execution and Generation

Generation requests are handled through the `batch()` method of the `Model` class to abstracts the differences between imported and API-based models. Prompts are appended to the conversational history and submitted to the model-specific `generate()` function. As stated before, this design makes the generation logic fully **modular** and **extensible**: the pipeline can switch between imported and API-based models without changing downstream processing; integrating a new model only requires supplying:

- **Imported models**: a function that, given a model and a tokenizer, return a function to batch a chat, returning a string as in Listing 5.5;
- **API models**: a function that, given a model, return a function to batch a chat, returning a string as in Listing 5.6.

Listing 5.5: Generate function for import models.

```
def new_model_generating_function_from_model(
    model_to_use) -> Callable[[List[str]], str]:

    def new_model_generating_function(chat):
        ...
        output_generated = model_to_use.batch(chat) #
            Substitute specific generating function here
        ...
        return output_generated

    return my_new_model_generating_function

match name:
    case 'my-new-model-name':
        model_to_use = load(model, tokenizer)
        return my_new_model_generating_function(
            model_to_use)
```

Listing 5.6: Generate function for APis models.

```
def load_generate_api_function(name, model, config,
    debug_print) -> Callable[[List[str]], str]:
    def my-new-model-generating-function(chat):
        ...
        output_generated = model.batch(chat) #
            Substitute specific generating function here
        ...
        return output_generated

    match name:
        case 'my-new-model-name':
            return my-new-model-generating-function
```

By using this unified approach, the pipeline maintains a consistent execution logic, ensuring that all models, regardless of their type, can be invoked seamlessly within the same workflow.

5.2.3 Prompting and Execution Time Measurement

Prompting is implemented ensuring consistency across models while remaining extensible. Each prompt is represented as a `list` of dictionaries, where every entry specifies a `role` (e.g. `system`, `user`, `assistant`) and its corresponding `content`. This role-based format guarantees compatibility with both imported and API-based models, as it mirrors the structure required by most modern chat interfaces.

For extensibility, the system allows either the use of a **base prompt**, shared across all models, or a **model-specific prompt**, defined in the configuration files. This design makes it possible to adopt default templates for general experiments while still accommodating provider-specific or fine-tuned prompt structures when needed.

Execution times are recorded for each generation request. Measurement is handled uniformly in the pipeline by marking the wall-clock time at the start and end of the request:

- for imported models, times are measured separately for CPU and GPU executions;
- for API models, times correspond to the full round-trip of the HTTP request.

Listing 5.7: Prompt and execution time measurement example.

```
# Load prompts for specific model and exercise
def load_prompts(prompt_version, model_name, exercise):

    # If model specific prompt is not found, the base
       one is loaded
    prompts = load_prompt(prompt_version, model_name)

    # Last prompt is the exercise text
    prompts.add(load_text_exercise(exercise))

    return prompts

prompts = load_prompts(prompt_version, model_name,
   exercise)
model.refresh_session()

# Time measurement
start_time = time.time()
model_output = model.batch(prompts)
end_time = time.time()
elapsed = end_time - start_time
```

All execution times are then stored alongside outputs and evaluation metrics in CSV format.

5.2.4 Output Parsing and Post-Processing

Model outputs are first collected and, whenever possible, parsed into `YAML` structures to enforce a standardized representation across experiments. If parsing fails, raw strings are preserved to avoid loss of information.

A dedicated post-processing stage then applies configurable rules as reported in Section 4.1.5. To ensure standardization in evaluation, the same post-processing rules are applied both to generated outputs and to ground-truth references. Finally, processed outputs are stored in `CSV` and `YAML` formats together with execution metadata, enabling reproducible analysis and external inspection.

5.2.5 Comparison and Metrics Evaluation

Once outputs and ground-truth references have been post-processed, they are compared to compute both standard and task-specific metrics. The evaluation considers both *nodes* (i.e. the set of fact, measures and dimension attributes) and *edges* (i.e. the directed "from → to" dependencies), thus capturing correctness at multiple levels of the generated schema.

For each level, the following notions are applied:

- **True Positives (TP)**: nodes or edges that appear in both the model output and the ground truth;
- **False Positives (FP)**: nodes or edges generated by the model but absent in the ground truth;
- **False Negatives (FN)**: nodes or edges that appear in the ground truth but are missing from the model output.

From these quantities, the standard metrics are derived:

- **Precision**: $\frac{TP}{TP+FP}$ measuring how many of the generated elements are actually correct;
- **Recall**: $\frac{TP}{TP+FN}$ measuring how many of the ground-truth elements have been correctly recovered;
- **F1-score**: $2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$ is the harmonic mean of precision and recall, balancing completeness and correctness. It's commonly used for graph comparison [18].

All metrics are computed separately for nodes and edges, providing a fine-grained view of model performance. In addition, **custom error metrics** are defined through domain-specific rules to capture discrepancies not fully explained by standard precision/recall-based measures.

Custom Errors

In addition to standard evaluation, the system defines a set of *custom errors* tailored to the domain of conceptual multidimensional modeling. These errors capture discrepancies that may not be adequately reflected by generic metrics but are crucial for assessing schema correctness. The custom error categories are:

- **Attributes**:
  - `shared_extra`: attributes that appear in both fact and dimension hierarchies in the output but not in the ground truth;
  - `shared_missing`: attributes that should appear in both contexts according to the ground truth but are missing in the output;

  - `shared_with_fact_root_extra`: attributes incorrectly connected to the fact root as well as to dimensions;
  - `shared_with_fact_root_missing`: expected attributes connected to the fact root are absent.

- **Dependencies**:
  - `extra`: dependency edges that exist in the output but not in the ground truth;
  - `missing`: dependency edges that are present in the ground truth but absent in the output;
  - `reversed`: dependency edges whose direction has been inverted compared to the ground truth.

- **Fact**:
  - `false_fact`: cases where an incorrect fact node has been introduced;
  - `incorrect`: a boolean flag signaling whether the chosen fact does not match the expected one.

- **Measures**:
  - `extra`: additional measures incorrectly introduced by the model;
  - `missing`: ground-truth measures that are absent in the model output.

- **Miscellaneous**:
  - `extra_disconnected_components`: disconnected subgraphs produced in the output, indicating structural errors in schema connectivity;
  - `extra_tags`: presence of invalid or unexpected YAML tags in the output.

By explicitly tracking these error categories, the evaluation provides a finer diagnostic of model behavior, highlighting not only whether the generated schema is correct, but also in which specific aspects it deviates from the intended conceptual model.

Finally, results are aggregated across runs and exported in `CSV` format, ensuring comparability across models, prompt configurations and execution environments.

MetricsCalculator

The `MetricsCalculator` class provides the foundation for computing standard evaluation metrics. It preprocesses ground-truth and generated outputs into normalized representations of facts, measures and dependencies. From these structures, it extracts true positives, false positives and false negatives at the node and edge level. The class also computes precision, recall and F1.

ErrorDetector

While the `MetricsCalculator` covers standard metrics, the `ErrorDetector` class is responsible for applying domain-specific validation. It builds on the indexes computed by the `MetricsCalculator`, detecting issues detailed in Section 5.2.5.

This two-level structure decouples generic metric computation from domain-specific validation, ensuring that the evaluation framework is both reusable and extensible.

Output obtained


```yaml
errors:
- attributes:
    shared_extra: 0
    shared_missing: 0
    shared_with_fact_root_extra: 0
    shared_with_fact_root_missing: 0
  dependencies:
    extra: 0
    missing: 1
    reversed: 0
  fact:
    false_fact: 0
    incorrect: false
  measures:
    extra: 1
    missing: 0
  miscellaneous:
    extra_disconnected_components: 0
    extra_tags: false
metrics:
- edges:
    f1: 0.9474
    fn: 1
    fp: 0
    precision: 1.0
    recall: 0.9
    tp: 9
  nodes:
    f1: 0.9091
    fn: 1
    fp: 1
    precision: 0.9091
    recall: 0.9091
    tp: 10
```


Listing 5.3: Output obtained after comparison.

5.2.6 Results Storage

The evaluation pipeline records all computed metrics and custom errors into structured `CSV` files. Each row corresponds to a single experiment run and includes identifiers such as the model name, configuration parameters, prompt variant and execution environment and time. Storing results in `CSV` format makes them directly usable for post-hoc analysis. Metrics can be aggregated across multiple runs, grouped by experimental factors, or visualized through plots and dashboards to identify performance trends.

5.3 Aggregation and Visualization

A dedicated aggregation script further processes the `CSV` files by applying filters (e.g., selecting specific models, exercises, or prompt variants) and consolidating results into unified datasets. These aggregated datasets are then used to generate the visualizations introduced in Section 4.2.

5.3.1 Graph drawing

To support debugging and qualitative inspection of the results, a dedicated JavaScript implementation was developed for graph visualization. This script takes as input both the model-generated output and the corresponding ground-truth schema, and produces side-by-side visual representations of their graphs. Each schema is rendered as a directed graph where nodes represent facts, measures, or attributes, while edges correspond to dependencies.

The script additionally highlights discrepancies between the two graphs, such as:

- **True positives**: nodes and edges correctly generated by the model;
- **False positives**: extra nodes or edges present only in the generated output;
- **False negatives**: nodes or edges missing from the generated output but present in the ground truth.

By visually overlaying these elements, the script provides an intuitive understanding of how metrics like precision, recall and F1 are reflected in practice. This visualization complements the numerical evaluation by enabling a more immediate identification of systematic errors (e.g., recurring missing dependencies or role misassignments), thereby streamlining the debugging process. An example of visual output is supplied in Figure 4.5.

5.4 GPU execution

Running experiments on GPU required the use of the university's computing infrastructure, managed through `Portainer`. Access to the GPU nodes was provided by containerized environments, meaning that the project had to be fully **dockerized**. A dedicated `Dockerfile` was prepared to ensure reproducibility, installing all required dependencies (Python, PyTorch with CUDA support, Hugging Face `transformers` and auxiliary libraries). This setup allowed the pipeline to be executed seamlessly on GPU nodes without manual configuration of the environment.

An excerpt of the `Dockerfile` is reported in 5.4.

```
# Docker image to enable cuda execution
FROM nvidia/cuda:12.2.2-cudnn8-runtime-ubuntu22.04
# Install system dependencies and python deadsnakes PPA
RUN apt-get update && apt-get install -y ...
# Install Poetry
RUN curl -sSL https://install.python-poetry.org | python3 - ...
# Set working directory
WORKDIR /app
# Copy project files (first pyproject to leverage caching)
COPY pyproject.toml poetry.lock* ./
# Install deps
RUN poetry install --no-root
# Copy rest of the project
COPY . .
```

Listing 5.4: DockerFile excerpt.

To streamline execution, automated scripts has been provided to run the program within `Portainer`, minimizing manual setup effort. For reproducibility and convenience, the same routines were also exposed as `Poetry` tasks, allowing experiments to be launched either directly from the development environment or through the containerized infrastructure with a single command.

5.5 Methodologies

This section outlines the methodological choices that guided the development process. The focus was on ensuring **automation** and **reliability**: a clear Git workflow supported structured contributions, continuous integration pipelines automated validation and release, while systematic testing ensured robustness and compliance with requirements. Together, these practices provided a disciplined framework for managing both code quality and experimental reproducibility.

5.5.1 Git Workflow

The project adopted a lightweight Git workflow based on **conventional commits**. Each feature or fix was developed in a dedicated branch, which was merged into the `develop` branch once completed. When a full functionality or fix was achieved, a pull request was opened to merge into `master`, ensuring code stability and traceability.

5.5.2 Continuous Integration

A CI/CD pipeline was implemented using GitHub Actions. It included stages for secret detection, semantic version computation, dependency installation, testing and conditional release with `semantic-release`, triggered on pushes and pull requests. This setup ensured that every change was validated, versioned and released in a controlled workflow.

5.5.3 Test

Testing played a central role in validating both functionality and business requirements. Tests were designed to detect errors early and to guarantee that new changes would not break previously implemented features. Core components under test included:

- Custom error detection functionalities;
- Metrics evaluation logic;
- Post-processing and rule-based equivalence checks;
- YAML parsing and schema validation.

This systematic testing provided confidence in the robustness and reliability of the implementation.

Chapter 6

Experimental evaluation

This chapter presents the experimental setup and results obtained from evaluating the proposed system. The goal of the experiments is to assess the ability of Large Language Models to generate Dimensional Fact Model schemas under both supply-driven and demand-driven design approaches. Performances are analyzed with respect to standard metrics, custom error tracking as well as comparing execution efficiency across CPU, GPU and API-based runs.

6.1 Experimental Setup

6.1.1 Tasks and Inputs

The set of test cases covers a diverse **range of domains**, each characterized by different levels of structural and semantic complexity. Table 6.1 reports their main features. The first column indicates the number of tokens in the corresponding natural language requirements, providing a measure of textual complexity. The remaining columns capture schema-related challenges:

- **Hidden functional dependencies**: dependencies not explicitly represented in the schema but implied by the semantics of the domain. Their detection is crucial for generating correct dimension hierarchies and relationships, yet LLMs may fail to infer them without explicit cues;

- **Composite foreign keys**: when foreign keys are composed of multiple attributes, they introduce additional complexity in recognizing relationships. This often leads to confusion in schema generation, as models must correctly group attributes and avoid misinterpreting them as separate links;

- **Ambiguity in fact identification**: some schemas allow multiple candidate facts. Selecting the wrong fact table results in incomplete or misleading dimensional schemata. This challenge tests the model's ability to reason about granularity and domain semantics beyond syntactic structure;

- **Shared hierarchies**: dimensions may share attributes or hierarchical paths (e.g. "date" used in different contexts, for both shipment and delivery). Models must reconcile these overlaps without creating redundancy, which requires deeper semantic alignment;

- **Cycles**: cyclic relationships in the source schema create difficulties in extracting a tree-like DFM structure. Identifying and resolving cycles into meaningful hierarchies or facts is particularly challenging, even for expert designers, and pushes the limits of automated schema generation.

Such variability ensures that the evaluation spans both simpler schemas and more intricate scenarios, enabling a deeper **analysis of the robustness and generalization capacity** of the tested models.

For each exercise, both supply-driven and demand-driven design approaches were systematically tested; within each approach, multiple prompt variants were explored, described in Section 4.1.3. This allowed the comparison of robustness of different prompting strategies across heterogeneous exercises and modeling settings.

Table 6.1: Test cases used for the experiment.

Id	Domain	# Tables/Attr.	# Tokens	Hidden FDs	Composite FKs	Fact ambiguity	Shared Hier.	Cycles
C1	Sw installations	5/14	49	N	N	N	N	N
C2	Purchases	5/19	54	Y	N	N	N	N
C3	Card purchases	5/34	94	Y	N	N	Y	N
C4	Car races	7/33	65	N	Y	N	N	N
C5	Crossfit	7/30	71	Y	Y	Y	N	N
C6	Staff recruitment	6/21	81	N	N	N	Y	Y
C7	Car rentals	8/35	106	Y	Y	Y	Y	N
C8	Train trips	9/38	130	Y	Y	Y	Y	N
C9	Flights	10/50	179	Y	Y	Y	Y	N

6.1.2 Models Tested

The following models were considered for evaluation, imported ones are accessed via Hugging Face, tested on CPU and GPU:

- **LLaMA variants Imported**: `llama-3.1-8B-inst`, `llama-3.2-1B-inst` and
`llama-3.2-3B-inst`;

- **Mistral Imported**: `mistral-7B-inst-v0.3`;

- **Falcon Imported**: `falcon-3-10B-inst`;
- **API-based models**: GPT-4 and GPT-5.

The choice of models followed an incremental strategy: larger and more recent versions (e.g. LLaMA 3.1–8B, LLaMA 3.2–3B) were tested alongside smaller, resource-constrained versions (e.g. LLaMA 3.2–1B). This allowed the comparison of efficiency and accuracy trade-offs across scales, balancing computational cost with model quality. Other architectures, including non-instruction-tuned variants, were explored but discarded as they either failed to produce parsable outputs or exceeded memory constraints.

6.1.3 Execution Environment

All experiments involving **imported models** were executed on the following hardware:

- **CPU execution**: `Intel(R) Xeon(R) w5-3425 @ 3.19 GHz`, equipped with 128 GB RAM;
- **GPU execution**: `NVIDIA RTX 6000 Ada Generation`, with 48 GB VRAM.

6.1.4 Metrics

The following metrics are evaluated:

- **Precision, Recall, F1-score** for nodes and edges
- **Custom error types**, identified through post-processing rules
- **Execution time**, recorded separately for CPU, GPU and API models

6.1.5 Evaluation Procedure

All experiments were automated to ensure reproducibility and consistency. Each model–prompt configuration was executed **ten times**, in order to account for the inherent non-determinism of LLM outputs. Results from repeated runs were then aggregated, exported in `CSV` format and further analyzed through the visualization procedures described in the previous sections.

6.2 Results

6.2.1 Per-Prompt Analysis

The per-prompt analysis has been initially conducted using the **GPT-4** model only. This choice allowed to focus on understanding the relative effectiveness of different prompt formulations in both supply-driven and demand-driven settings. The insights gained from this analysis guided the selection of the most promising prompts, which were later adopted and tested across other models in order to perform a broader evaluation in terms of both *effectiveness* and *efficiency*.

Ground truth definition

For **supply-driven** design, the ground truth was generated by applying the **FD-chasing algorithm** proposed in [5] to the source schema; the fact was then selected as the source table maximizing coverage of attributes in the DFM schema. Since the FD-chasing algorithm is deterministic and, within these test cases, no ties occur in fact selection, exactly one ground truth is determined for each case. For **demand-driven** design, the **ground truth was created manually by a domain expert**, starting from the requirements.

Name-matching and count measures

A central issue in metric computation is the matching of schema element names. In supply-driven design, exact matching is enforced as `RELATION.Attribute`, while in demand-driven design the model may use semantically correct but lexically different names. To address this, for each test case a list of equivalent names has been made (e.g. *CreditCardType* and *CardType*). Another issue is the presence of count measures: in principle, these should only appear when fact granularity is coarser than the source database, yet inferring granularity from textual requirements is non-trivial even for human designers. Therefore, all count measures were removed from demand-driven ground truths, and any such measures generated by the models were ignored in evaluation.

Results with basic prompts

With the first basic **rq2** prompt for supply-driven approach, the average F1-scores of nodes and arcs is ≈ 0.41, denoting a **poor** performance. The most frequent errors include:

- **Missing FDs**;
- **Reverse FDs** as arcs generated in the wrong direction;

- **Wrong fact selection**;
- **Missing measures**;
- **YAML syntax errors** as invalid tags or indentation.

Improved supply-driven prompts

An improved version of the basic prompt has been investigated provided in two formulations: an *algorithmic* and a *declarative* variant. The average F1-scores of nodes and arcs for the algorithmic version is ≈ 0.81 and ≈ 0.78 for the declarative one. Performances with these prompts can be classified as **good**. Residual errors mostly concern missing arcs, leading to fragmentation and missed shared hierarchies. Fact selection is generally correct, except for some runs of C5 and systematically in C7.

Demand-driven prompts

With the base prompt **rq4**, average F1-scores of nodes and arcs is ≈ 0.75, leading to an **average** classification. Common errors include:

- **Reverse or missing FDs**;
- **Wrongly identified dimensions**;
- **Non-recognition of shared hierarchies** (e.g. *shipment date* vs. *delivery date*).

Improved demand-driven prompts

When improved to **rq5**, performance becomes excellent, with average F1-scores of ≈ 0.91, considered as **excellent** performances. Beyond error reduction, the model occasionally introduces useful additional concepts not explicitly present in requirements (e.g. *session* in C5). Remaining issues concern shared hierarchies and unique identifiers.

Overall, these findings confirm that **careful prompt engineering significantly boosts accuracy in both supply-driven and demand-driven design**, shifting performance from poor/average to good/excellent.

6.2.2 LLM Effectiveness evaluation

To evaluate the effectiveness of large language models in conceptual design tasks, a dedicated set of experiments was performed using **GPT-4**.

Technological Setup

The GPT-4 experiments were executed through the OpenAI API, accessed via the Azure distribution. The configuration parameters used are reported in 6.1.

```yaml
model_api:
  name: 'gpt'
  label: 'gpt4'
  deployment: 'gpt-4o'
  api_version: '2024-04-01-preview'
  max_tokens:
  n_responses: 1
  temperature: 0.1
  stop:
  top_p: 0.9
```

Listing 6.1: Configurations for GPT experiments.

Results Graphs

Graphs are organized per prompt type, for each case both **nodes** and **edges** metrics are shown.

- **Supply-driven prompts:** RQ2, RQ3-dec, RQ3-alg;
- **Demand-driven prompts:** RQ4, RQ5.

Both the followings Figure 6.4 and Figure 6.7 present, for each prompt, the average F1 and time across all test cases.

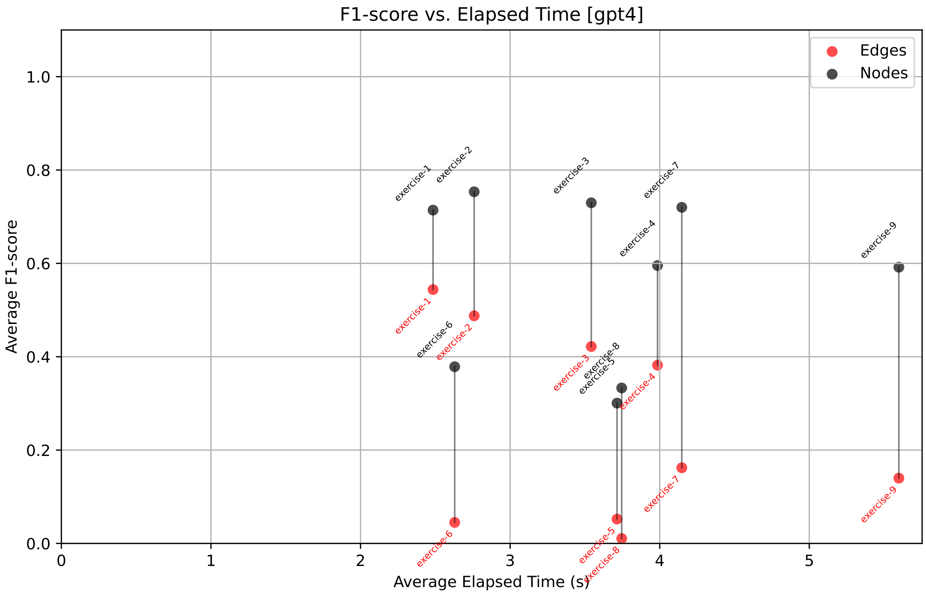


Figure 6.1: GPT4 supply-driven output graph – RQ2.

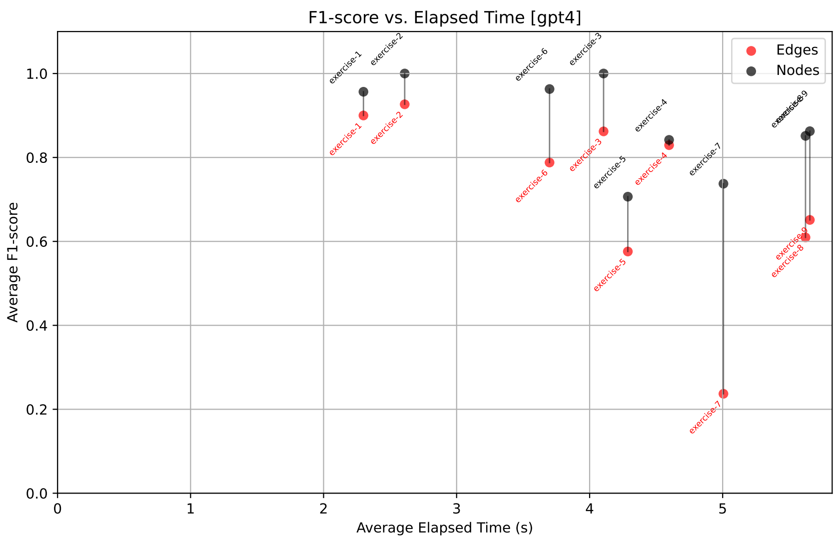


Figure 6.2: GPT4 supply-driven output graph – RQ3-ALG.

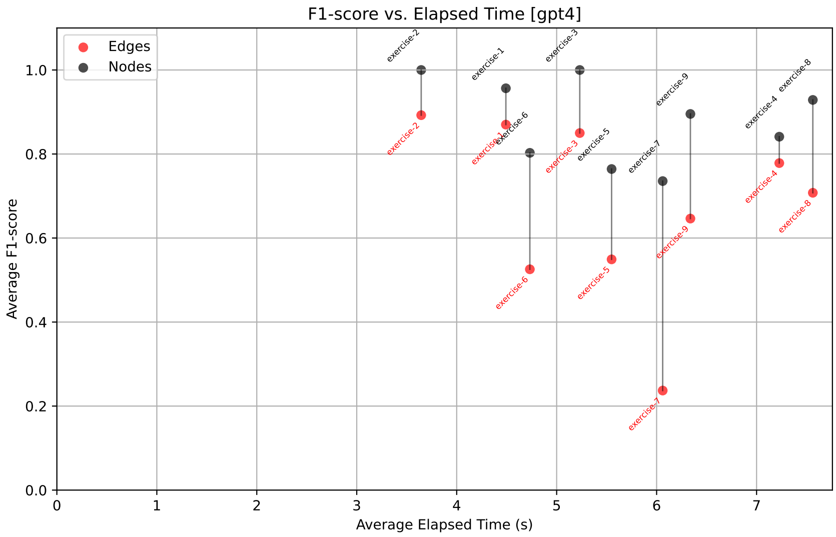


Figure 6.3: GPT4 supply-driven output graph – RQ3-DEC.

Figure 6.4: GPT4 supply-driven output graphs for each prompt version.

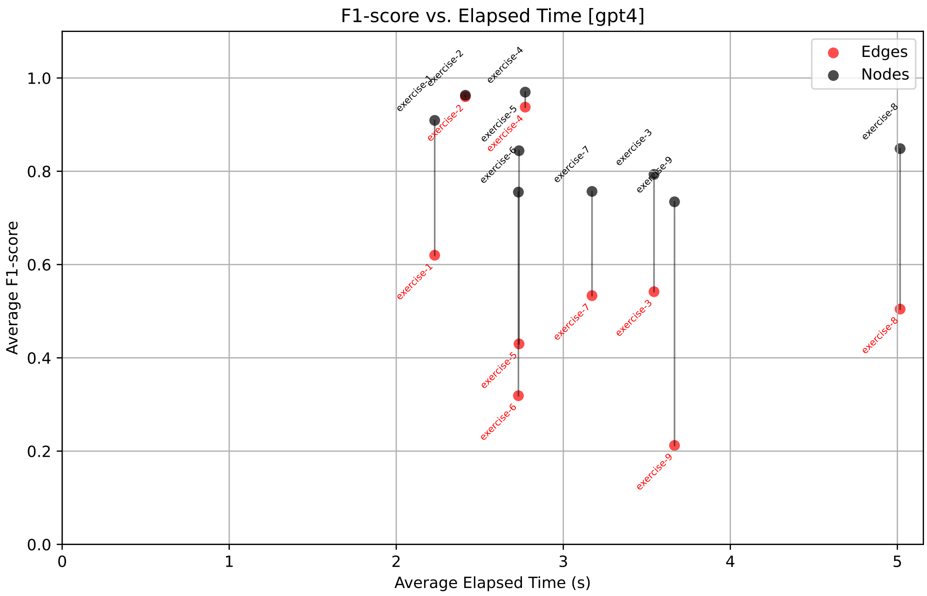


Figure 6.5: GPT4 demand-driven output graph – RQ4.

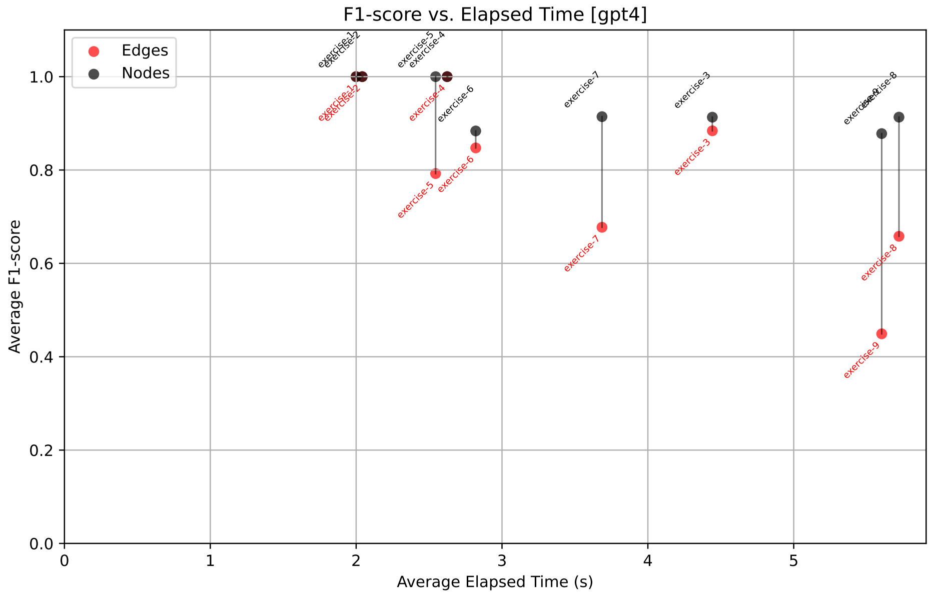


Figure 6.6: GPT4 demand-driven output graph – RQ5.

Figure 6.7: GPT4 demand-driven output graphs for each prompt version.

Results Overview

Effectiveness was assessed across the full set of exercises C1–C9, considering all prompts. Results show a clear progression depending on the prompt design, with a summary in Table 6.2:

- **RQ2 basic supply-driven**: average F1 of ≈ 0.41, rated as *poor*, with frequent errors in FD recognition, fact selection and YAML syntax;
- **RQ3 improved supply-driven**: with declarative and algorithmic variants, average F1 improved up to ≈ 0.81, rated as *good*. Errors were reduced to mainly missing arcs and non-recognized shared hierarchies;
- **RQ4 basic demand-driven**: average F1 of ≈ 0.75, rated as *average*, with issues in FD directionality, wrong dimensions and missed shared hierarchies;
- **RQ5 improved demand-driven**: average F1 of ≈ 0.91, rated as *excellent*. Errors were minimal, mainly related to shared hierarchies, while in some cases GPT-4 introduced useful additional concepts beyond the ground truth.

Table 6.2: Average F1 scores of nodes and arcs for the different RQs.

Approach	**RQ**	**F1-nodes**	**F1-arcs**	**F1-avg**
Supply-driven	RQ2	0.57	0.24	0.41
	RQ3-alg	0.90	0.72	0.81
	RQ3-dec	0.88	0.67	0.78
Demand-driven	RQ4	0.86	0.64	0.75
	RQ5	0.96	0.85	0.91

Discussion

Overall, GPT-4 demonstrated a strong capability in conceptual design tasks, provided that prompt engineering is carefully applied. The results highlight that:

1. Basic prompts (RQ2, RQ4) yield only partial correctness, with high variability across exercises;

2. Improved prompts (RQ3, RQ5) significantly enhance both precision and recall, consistently producing near-complete schemata;

3. **Prompt engineering** can thus be considered the **main driver of effectiveness**, with GPT-4 adapting well to both supply-driven and demand-driven designs.

Results with GPT-5

At the time of the initial experiments, GPT-5 was not yet publicly available. Following its release, we replicated the same set of tests conducted with GPT-4 to evaluate potential improvements in effectiveness.
Overall, GPT-5 achieved **slightly better** performance across both supply-driven and demand-driven prompts. The improvements are most evident in edge-level metrics, while node-level performance remains largely comparable to GPT-4. This indicates that the newer model version is more effective at capturing relational dependencies, while preserving the same reliability in identifying schema elements. A detailed comparison is reported in Figure 6.8, where average F1 scores and execution times across all exercises are aggregated by prompt type. In particular, GPT-5 exhibits a little lower execution times and slightly better effectiveness across all prompts, with the only exception being RQ4, where its performance marginally lags behind GPT-4.

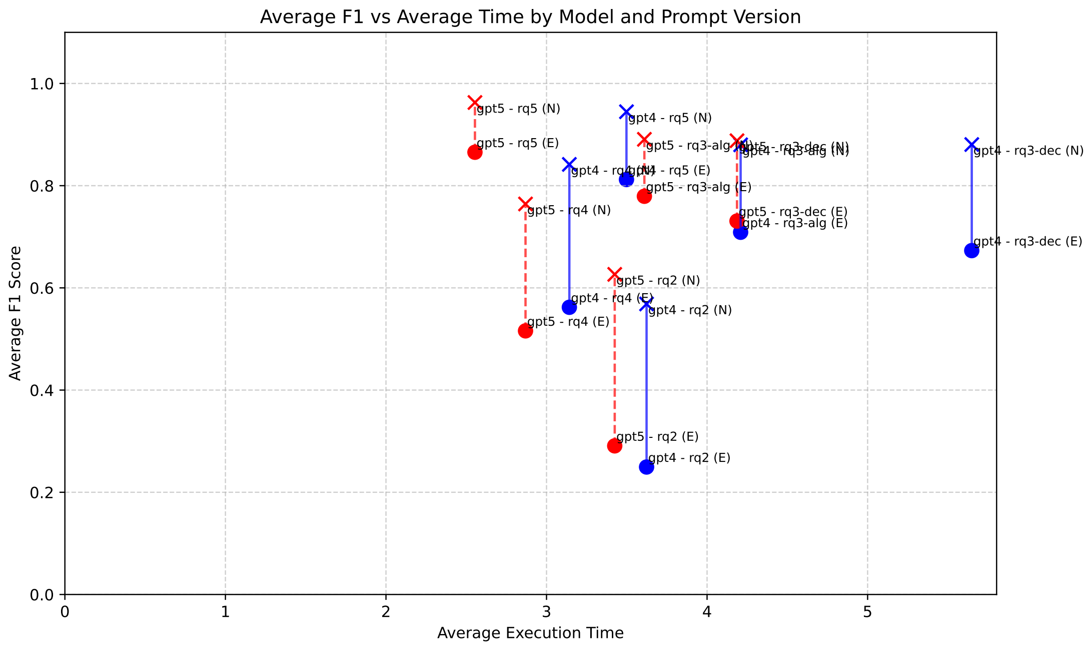


Figure 6.8: Comparison of GPT-4 and GPT-5 by prompt versions.

6.2.3 LLM Efficiency evaluation

To evaluate the efficiency of locally imported LLMs, tests has been conducted on both CPU and GPU execution environments. Each model was evaluated on all exercises, reporting execution time and average F1 score as mean of nodes and edges. The boxplot in Figure 6.9 shows execution time on the Y-axis and models on the X-axis, sorted by average F1, thus highlighting the trade-off between speed and accuracy for CPU and GPU runs. A boxplot was chosen because, especially in CPU execution, times exhibit strong variability across runs: this representation effectively captures both central tendency and dispersion.
The experiments reported correspond to 10 runs for each prompt, restricted to the most promising ones **rq3-dec** and **rq5**, thereby mixing supply and demand-driven approaches. Among the tested models, **Falcon3** achieved the highest average F1 score across nodes and edges.
From an efficiency perspective, GPU execution provides a remarkable speed-up compared to CPU and, more importantly, much greater stability across runs. CPU boxplots are considerably longer, with outliers producing both unusually high and unusually low times relative to the average, while GPU times are consistently compact. Interestingly, execution times do not differ substantially across models on the same hardware, suggesting that resource utilization is relatively uniform

once the environment is fixed.
Comparing these results with the overall effectiveness analysis, imported models such as Falcon3 demonstrate competitive trade-offs, but **GPT-based models** remain by far the most accurate, consistently achieving higher F1 scores. Efficiency gains from GPU usage are thus shared across models, but the **gap in effectiveness clearly favors GPT**, making it the strongest option when accuracy is the primary concern.

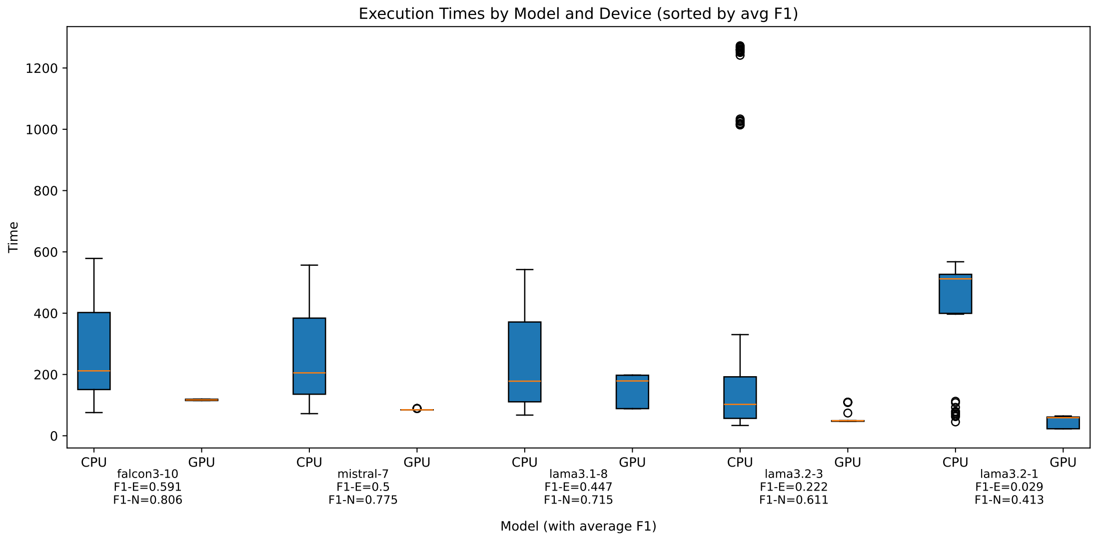


Figure 6.9: Execution time comparison of imported models on CPU and GPU. Models are ordered by average F1 score across nodes and edges.

6.3 Discussion

The experimental results provide several insights into the relative performance of models, execution settings and evaluation procedures.

Suitability of Models for Supply-Driven vs Demand-Driven Design

The experiments highlight clear differences between approaches. Imported models (e.g. Falcon3, LLaMA variants, Mistral) show reasonable performance in demand-driven tasks, particularly when prompts are carefully engineered, but struggle with supply-driven design due to their limited ability to detect functional dependencies and consistently choose the correct fact. Conversely, GPT-based models (GPT-4 and GPT-5) demonstrate stronger adaptability to both approaches, with GPT-5 showing a slight improvement over GPT-4 in capturing relational dependencies at the edge level. Prompt engineering proves decisive: basic prompts yield poor accuracy in supply-driven design, while improved prompts (rq3-dec, rq5) allow GPT models to reach good to excellent results.

Impact of Execution Environment

Execution environment plays a major role in efficiency but not in accuracy. Imported models executed locally benefit significantly from GPU acceleration, which reduces execution time by up to an order of magnitude and stabilizes variance across runs. CPU runs are much slower and highly variable, with frequent outliers. However, execution times across different imported models remain comparable once hardware is fixed. On the other hand, API-based models abstract away hardware concerns, with latency dominated by API call and network response. Despite this, their effectiveness in schema generation far exceeds that of imported models, making them the most reliable choice when accuracy is prioritized over execution time.

Role of Post-Processing Rules

Hand-crafted post-processing rules play a crucial role in aligning LLM outputs with the expected ground truths. In particular, they resolve name-matching issues in demand-driven design, normalize measure treatment and enforce YAML syntax validity. Without such rules, metrics would underestimate model performance, especially for demand-driven outputs where naming variability is natural. Nonetheless, this reliance on manual alignment reduces portability and generalizability to other domains.

Limitations in LLM Schema Generation

The experiments confirm both strengths and weaknesses of LLMs in conceptual schema design. GPT-based models achieve high accuracy with appropriate prompting, but still exhibit residual errors such as missing arcs, misidentified dimensions, or overlooked hierarchies. Imported models, while more efficient under GPU acceleration, are limited in memory capacity and produce lower-quality outputs, especially for complex exercises. Finally, all models operate under a single-turn assumption, whereas real-world schema design is inherently iterative and interactive.

Unparsable Responses from Imported Models

Despite the structured prompt design and the post-processing pipeline, imported models occasionally produced unparsable outputs, mainly due to invalid YAML syntax or incomplete generations. While these cases were infrequent, they highlight an additional limitation of locally run models compared to GPT-4 and GPT-5, which consistently returned syntactically valid responses. Such failures required discarding runs or applying manual corrections, slightly reducing the robustness of the evaluation. To mitigate this problem, coding algorithmic parts of the task into the pipeline and introducing training integrated with process-based supervision may be effective, as explored in [14, 23].

Chapter 7

Conclusions

This thesis has investigated the application of Large Language Models to the generation of Dimensional Fact Model schemas. The study focused on both methodological and experimental aspects, with the aim of evaluating the effectiveness and efficiency of different LLMs under supply-driven and demand-driven design paradigms.
The main contributions can be summarized as follows:

- **Pipeline Design and Implementation**: a modular evaluation pipeline was developed, integrating model configuration, generation, post-processing and automated metric computation. This ensured repeatability, extensibility and facilitated large-scale experimentation;
- **Model Abstraction**: a unified `Model` class in Python was introduced to seamlessly handle both imported Hugging Face models and API-based models, enabling consistent experimentation across CPU, GPU and API execution environments;
- **Evaluation Framework**: a standardized set of post-processing rules was proposed, together with domain-specific error metrics, allowing fair and fine-grained comparison between model outputs and reference ground truths;
- **Visualization**: results were aggregated and reported through comprehensive plots (F1 and execution time trade-offs), providing intuitive insights into model behavior and comparative trends.

Overall, this work demonstrates the potential of LLMs as assistants in conceptual schema design, while also stressing the importance of prompt engineering, post-processing and systematic evaluation. The contributions provide a foundation for reproducible experimentation, model comparison and methodological reflection, paving the way for further research and more advanced applications in the field.

Overall, this work demonstrates the potential of LLMs as assistants in conceptual schema design, highlighting the importance of prompt engineering, post-processing, and systematic evaluation. These contributions not only establish a foundation for reproducible experimentation and rigorous model comparison, but have also been leveraged in [16], paving the way for further research and more advanced applications in the field.

7.1 Discussion of Findings

The experimental evaluation provides several key insights:

- **Relative performance of models**: API-based models such as GPT-4 and GPT-5 consistently outperform imported open-source models in both precision and recall, particularly under the demand-driven approach. Imported models can still deliver reasonable accuracy but remain less robust on complex exercises;

- **Impact of execution environment**: Execution times vary significantly across CPU, GPU and API runs. GPU-based inference enables competitive local execution with stable runtimes, while CPU runs show strong variability. API calls deliver superior accuracy, making them a solid choice for scenarios requiring both high accuracy and reasonable speed;

- **Role of post-processing**: Post-processing rules are critical for aligning raw model outputs with evaluation expectations. They mitigate frequent errors such as inconsistent naming, redundant attributes or structural omissions and directly influence reported metrics;

- **Trade-offs between efficiency and accuracy**: Lightweight imported models highlight the tension between computational cost and performance. While they can run efficiently on commodity hardware, they fall short of API-based models in handling complex schema generation tasks.

These findings underline the importance of prompt engineering and post-processing in improving results, while also clarifying the practical boundaries of current LLMs: high-performing API models set the accuracy benchmark, but their use comes with potential downsides such as vendor lock-in, data privacy concerns and ongoing usage costs, motivating further exploration of optimized or fine-tuned local alternatives.

7.2 Limitations

Despite the contributions, some limitations emerged during the study:

- The evaluation is constrained to a **finite set of exercises**, which may limit generalizability to other domains or schema complexities;
- **Post-processing rules** are **hand-crafted and context-dependent**, potentially reducing portability and requiring adaptation when applied to different datasets or requirements;
- The system assumes a **single-turn evaluation** for most runs, while real-world schema design often involves iterative refinement and feedback cycles;
- The usage of larger imported LLMs was limited by **memory constraints**, preventing testing of some high-capacity models that might have offered different performance trade-offs.

7.3 Future Work

Several directions can extend this research:

- **Expanded Dataset**: applying the pipeline to a larger and more diverse set of exercises to improve robustness of conclusions and evaluate performance on more complex schemas;
- **Automated Post-Processing**: developing machine learning or heuristic-based alignment strategies to replace manually defined rules, improving portability and reducing manual effort;
- **Interactive Evaluation**: extending the pipeline to simulate multi-turn schema design sessions, closer to real-world design practices;
- **Larger LLMs and Training Frameworks**: investigating high-capacity import models that were previously limited by memory, potentially providing **fine-tuning** or instruction-tuning pipelines to adapt these models to the DFM task, as well as injecting knowledge about multidimensional modeling into LLMs is another promising direction, as suggested by recent comparative studies on automated domain modeling [3];
- **Broader API Coverage**: extending experiments to other API-accessible models beyond GPT, enabling a comparative analysis across different architectures and vendor distributions.

Bibliography

[1] Syed Juned Ali, Iris Reinhartz-Berger, and Dominik Bork. How are llms used for conceptual modeling? an exploratory study on interaction behavior and user perception. In *International Conference on Conceptual Modeling*, pages 257–275. Springer, 2024.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[3] Kua Chen, Yujing Yang, Boqi Chen, José Antonio Hernández López, Gunter Mussbacher, and Dániel Varró. Automated domain modeling with large language models: A comparative study. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 162–172. IEEE, 2023.

[4] Hans-Georg Fill, Peter Fettke, and Julius Köpke. Conceptual modeling and large language models: impressions from first experiments with chatgpt. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 18:1–15, 2023.

[5] M Golfarelli and S Rizzi. Data warehouse design: Modern principles and methodologies 5 mcgraw-hill. *New York*, 2009.

[6] Matteo Golfarelli and Stefano Rizzi. A methodological framework for data warehouse design. In *Proceedings of the 1st ACM international workshop on Data warehousing and OLAP*, pages 3–9, 1998.

[7] Walid Hariri. Unlocking the potential of chatgpt: A comprehensive exploration of its applications, advantages, limitations, and future directions in natural language processing. *arXiv preprint arXiv:2304.02017*, 2023.

[8] William H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Hoboken, NJ, 2002.

[9] Petar Jovanovic, Oscar Romero, Alkis Simitsis, Alberto Abelló, and Daria Mayorova. A requirement-driven approach to the design and evolution of data warehouses. *Information Systems*, 44:94–119, 2014.

[10] Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling, 2nd Edition*. Wiley, 2002.

[11] Rainer Lutze and Klemens Waldhör. Generating specifications from requirements documents for smart devices using large language models (llms). In *International Conference on Human-Computer Interaction*, pages 94–108. Springer, 2024.

[12] Kevin Ma, Daniele Grandi, Christopher McComb, and Kosa Goucher-Lambert. Conceptual design generation using large language models. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 87349, page V006T06A021. American Society of Mechanical Engineers, 2023.

[13] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[14] Lukas Netz, Judith Michael, and Bernhard Rumpe. From natural language to web applications: Using large language models for model-driven software engineering. In *Modellierung 2024*, pages 179–195. Gesellschaft für Informatik eV, 2024.

[15] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[16] Stefano Rizzi, Matteo Francia, Enrico Gallinucci, and Matteo Golfarelli. Conceptual design of multidimensional cubes with llms: An investigation. *Data & Knowledge Engineering*, page 102452, 2025.

[17] Oscar Romero and Alberto Abelló. Data-driven multidimensional design for olap. In *International Conference on Scientific and Statistical Database Management*, pages 594–595. Springer, 2011.

[18] Jonathan Silva, Qin Ma, Jordi Cabot, Pierre Kelsen, and Henderik A Proper. Application of the tree-of-thoughts framework to llm-enabled domain modeling. In *International Conference on Conceptual Modeling*, pages 94–111. Springer, 2024.

[19] Nectaria Tryfona, Frank Busborg, and Jens G Borch Christiansen. starer: A conceptual model for data warehouse design. In *Proceedings of the 2nd ACM international workshop on Data warehousing and OLAP*, pages 3–8, 1999.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[21] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*, pages 71–108. Springer, 2024.

[22] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *CoRR*, abs/2303.18223, 2023.

[23] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

[24] Zhibin Zhou, Jinxin Li, Zhijie Zhang, Junnan Yu, and Henry Duh. Examining how the large language models impact the conceptual design with human designers: A comparative case study. *International Journal of Human–Computer Interaction*, 41(10):5864–5880, 2025.