POLITECNICO DI TORINO

Collegio di Ingegneria Gestionale e della Produzione

Corso di Laurea Magistrale in Engineering and Management



Large Language Models to Support Natural Language Interaction with Manufacturing Execution Systems

Supervisor
Prof.ssa Giulia Bruno

Candidate
Andrea Del Sante

A.A. 2025/2026

Contents

Abstract **4**

1 Introduction **6**

2 Theoretical background **9**
2.1 Industry 4.0 and Industry 5.0 9
2.2 Manufacturing Execution Systems 14
2.3 Artificial Intelligence and Large Language Models 17

3 State of the art **21**
3.1 Text-to-SQL and MES 21

4 Methodology **28**
4.1 Proposed Framework 28
4.2 Evaluation Methodology 33
4.3 Prompt Engineering 34
4.3.1 Step 1 — Baseline: dynamic schema, minimal rules (59% Accuracy) 36
4.3.2 Step 2 — Domain rules for aggregation and structure (72% Accuracy) 38
4.3.3 Step 3 — Ground-truth verified fixes (86%) 42
4.4 Experimental Design 47

5 Implementation **49**
5.1 run_evaluation_cycles.py 49
5.2 test_question.py 55
5.3 fashion.py 59
5.4 llm.py 61
5.5 Evaluation.py 65

6 Conclusions **69**
6.1 Summary of the Work 69
6.2 Key Aspects 70
6.3 Possible Future Work 72
6.4 Final Considerations 74

Abstract

The increasing adoption of Industry 4.0 and Industry 5.0 principles is reshaping the way manufacturing companies organise their production processes and the way their employees interact with information systems. In this context, improving the usability of manufacturing information systems becomes a strategic priority: enabling workers and managers to access production data intuitively and efficiently not only increases operational productivity but also improves working conditions.
Manufacturing Execution Systems represent the main repository of production knowledge, collecting and organising data. Traditional MES interfaces are based on predefined forms that require specialized training and limit the accessibility of information for non-technical users. The introduction of natural language interfaces powered by Large Language Models offers a concrete solution to this limitation, allowing any stakeholder, from shop floor operators to production managers, to query the system in their own language without any knowledge of SQL or the underlying database structure.
The objective of this work is to build a system that employs Large Language Models as its core engine to accurately translate natural language questions into executable SQL queries against a MES database. The system is designed to be easily adapted to other databases. Its performance is then evaluated across three state-of-the-art models, GPT-5 (OpenAI), Claude Opus 4 (Anthropic), and Gemini 2.5 Pro (Google), to verify whether it maintains its effectiveness when used with different models.
The evaluation is conducted on the benchmark proposed by Yuan et al. [1], consisting of 55 natural language requests paired with ground truth SQL queries on a garment MES database. Each model is evaluated over 10 independent cycles, for a total of 1,650 query evaluations. The prompt was developed through an iterative engineering process driven by systematic mismatch analysis, starting from a baseline execution accuracy of approximately 60% and reaching approximately 86% across all three models. The convergence of three fundamentally different model architectures to the same accuracy level demonstrates that with a sufficiently engineered domain-specific prompt, the

choice of model becomes secondary, and the prompt itself is the primary determinant of system performance.

Introduction

The digitalization of manufacturing has significantly accelerated over the past decade, driven by the adoption of Industry 4.0 principles and, more recently, by the human-centric concepts of Industry 5.0. Modern factories generate increasingly large volumes of data (containing production orders, order status, completed and partially completed operations, task assignments, material allocations, working group progress, quality metrics, and inventory levels) which are collected and managed by Manufacturing Execution Systems. These platforms have become indispensable tools for production management, providing real time visibility of the state of the factory floor and enabling data driven decision making at every level of the organization. In today's competitive landscape, the ability to make fast and informed decisions has become a critical success factor: companies that can access, aggregate and interpret production data quickly are better positioned to react and to make decisions rapidly, optimizing resource allocation and maintaining operational efficiency.

In this context, digitalization is no longer simply a technological upgrade, it is a strategic necessity, and the accessibility of information across all levels of the organization is one of its most important dimensions.

Despite their central role, MES platforms are often perceived as difficult to use. Their interfaces are typically designed around predefined forms, fixed dashboards and structured workflows that require specialized training. These platforms store an increasingly large amount of operational data, but data alone does not automatically translate into useful information. Without the ability to query, filter, and interpret it, raw data risks remaining inaccessible to people. The gap between the richness of the information stored in the system and the ability of non-technical users to extract it represents one of the most significant inefficiencies in modern manufacturing organizations.

The emergence of Large Language Models has opened a new avenue for addressing this problem. LLMs are capable of understanding natural language questions and generating structured outputs, including SQL queries. By placing an LLM-based Text-to-SQL interface between the user and the MES database, it becomes possible to allow any employee to query production data simply by typing a question in their own language, without requiring technical knowledge of SQL or the underlying database structure. This approach is fully aligned with the human-centric pillar of Industry 5.0: rather than requiring workers to adapt to the system, the system adapts to the worker.

This thesis work focuses on the practical need to make MES data more accessible through natural language. The starting point of this work is the Chat with MES framework proposed by Yuan et al. [1], which provided the database, the benchmark of 55 natural language requests and an initial implementation of the Text-to-SQL pipeline. Testing the available code on the benchmark revealed an execution accuracy of approximately 60%, which served as the baseline for this work. Starting from this baseline, the objective of this thesis is to design and implement a robust, domain-specific Text-to-SQL system capable of significantly improving this accuracy through iterative prompt engineering, without relying on additional pipeline components or specialized infrastructure.
A key element of the system is scalability, both with respect to the underlying database and with respect to the language model used. Rather than embedding a static description of the database schema in the prompt, the system queries the database structure at runtime, making it automatically adaptable to any schema change without manual intervention. Similarly, the system is independent of the model: the same prompt and pipeline are evaluated across three state-of-the-art models, GPT-5 (OpenAI), Claude Opus 4 (Anthropic), and Gemini 2.5 Pro (Google), to assess whether the approach maintains its effectiveness across different model architectures.
The structure of this thesis is outlined as follows. Chapter 2 introduces the theoretical foundations, addressing key concepts such as the concept of Industry 4.0 and 5.0, Manufacturing Execution Systems and the core concepts underlying artificial intelligence and Large Language Models. Chapter 3 reviews the state of the art in LLM-based natural language interfaces for databases, with particular attention to Text-to-SQL systems and their application in manufacturing contexts. Chapter 4 explains the research methodology, starting from the initial context and defining the objectives to be achieved. It introduces the database used in this work and describes the approach adopted to evaluate the performance of the system, including the

prompt engineering process, which was driven by the systematic analysis of failure cases identified during iterative evaluation cycles. Chapter 5 focuses on the implementation phase, providing a detailed description of the proposed pipeline. Chapter 6 presents the experimental results obtained across the three selected models, summarizes the key findings of the study and discusses potential directions for future research.

Theoretical background

2.1 Industry 4.0 and Industry 5.0

The concept of Industry 4.0 was first introduced in 2011 by a team of scientists developing a high-tech strategy for the German industrial policy. The vision rapidly spread beyond its original national context: in 2016, the World Economic Forum [1] adopted "Mastering the Fourth Industrial Revolution" as the theme of its annual meeting in Davos, bringing the concept to a global audience. [2]

The term follows the versioning logic of software development: the first industrial revolution was powered by steam, the second by electricity, the third by computers and automation and the fourth by the integration of cyber-physical systems and digital networks into manufacturing [3].
Unlike previous industrial revolutions, which were driven by a single dominant technology, Industry 4.0 is characterized by the convergence of multiple enabling technologies. The central element is the cyber-physical system (CPS): the integration of computational processes and physical machinery, in which embedded sensors, actuators and controllers continuously exchange data with digital models of the production environment. The concept of Industry 4.0 is represented by an interconnected system, that connects machines, methods, and products.

A useful framework for understanding the technological infrastructure of Industry 4.0 is the concept of the smart factory, which integrates three interconnected dimensions:

[1] The World Economic Forum (WEF) is an international non-governmental organization, which annually convenes political, economic and academic leaders in Davos to discuss the world's most relevant global challenges.

- **smart production**, encompassing advanced manufacturing technologies that enable collaboration between operators, machines and tools;
- **smart services**, referring to the digital and technical infrastructures that allow integration across internal systems and between external actors such as suppliers and customers;
- **smart energy**, addressing the optimization of energy consumption in line with the principles of sustainable production.

Industry 4.0 focuses on the establishment of intelligent products and production processes to face the need of rapid product development, flexible production and complex environments [4].
The technological foundation of this vision was systematically mapped in a report published by the Boston Consulting Group [5], which identified nine enabling technologies as the building blocks of Industry 4.0:

- **Advanced manufacturing solutions**: interconnected and modular production systems, including automated material handling and advanced robotics such as collaborative robots (cobots).
- **Additive manufacturing**: layer-based production technologies that improve material efficiency and enable the production of complex geometries.
- **Augmented reality**: vision systems that support operators in performing tasks by overlaying digital information on the physical environment.
- **Simulation**: real-time modeling of interconnected machines and processes to optimize production before physical execution.
- **Horizontal and vertical system integration**: seamless exchange of information across all actors and levels of the production process, from shop-floor devices to enterprise systems.
- **Industrial Internet of Things (IIoT)**: communication infrastructure connecting production elements both within and beyond the factory, enabled by internet protocols.
- **Cloud computing**: scalable platforms for data storage, processing and analytics, enabling access to large volumes of manufacturing data through open systems.

- **Cybersecurity**: protection of the information and control systems that are increasingly exposed by the growing density of internal and external interconnections.
- **Big data and analytics**: techniques for processing and extracting predictive insights from large and heterogeneous data streams generated across the production environment.

Four core design principles characterize Industry 4.0 systems [4]:

- **Interconnection**: machines, devices, sensors and people are connected and able to communicate with each other through the Internet of Things.
- **Information transparency**: sensor data from the physical world are used to create a digital representation of the factory, providing operators with comprehensive information to make informed decisions.
- **Technical assistance**: systems support humans in aggregating and visualizing information, performing tasks that are difficult or unsafe for humans and providing decision support.
- **Decentralized decisions**: cyber-physical systems are capable of making autonomous decisions and performing tasks with minimal human intervention.

In this context, the Manufacturing Execution System represents one of the central components of Industry 4.0 architectures. Manufacturing Execution Systems (MES) have evolved from basic production tracking tools into intelligent, AI-driven platforms integrated with IoT, cloud computing and cyber-physical systems. As production systems grow in complexity and generate increasing volumes of data, the ability to extract information from MES databases in real time becomes a critical competitive advantage.

However, data alone is not sufficient. Raw data must be transformed into accessible, interpretable information before it can support decision making. The increasing interconnection of production systems and the growing volume of data generated are valuable only as the right information can be retrieved at the right moment by the right person. In this sense, accessibility is as important as availability: a system that stores large amounts of data, but requires technical expertise to query it, creates an information bottleneck, limiting the ability of operators and managers to react quickly to production events. Reducing this bottleneck is one of the central challenges

of modern MES and it directly determines the capacity of an organization to make punctual decisions at every level of the production hierarchy.

It is in this context of reactivity and speed that Large Language Models, combined with Text-to-SQL systems, play a significant role. By enabling operators to interact with MES databases in natural language, these systems remove the technical barrier between the user and the information system, making production data accessible without requiring knowledge of SQL or the underlying database schema. This facilitates not only vertical communication, allowing managers and engineers to query aggregated production data without technical intermediaries, but also horizontal communication among operators on the shop floor, who can access and share operational information in an intuitive and immediate way, enabling real-time decision-making and operational efficiency at a scale that was not previously achievable [6].

While Industry 4.0 focused primarily on automation, efficiency, and machine communication and connectivity, the emerging framework of Industry 5.0, introduced by the European Commission in January 2021, centers the human being at the heart of the production system [7]. Industry 5.0 emerged as a conceptual extension, complementing its technological achievements with a wider set of social goals. If Industry 4.0 is primarily technology-driven, Industry 5.0 is value-driven: it places worker well-being, environmental sustainability and systemic resilience at the same level of priority as productivity and efficiency [8].

In this framework, technology is no longer considered merely as a means of automation and optimization, but as a tool to be implemented in relation to the human capital present in the organization. The goal is to enhance and support the work of human operators, enabling them to focus on those tasks where human judgment, creativity and adaptability provide added value that a machine cannot replicate with the same effectiveness. Rather than replacing the worker, technology is designed to augment human capabilities, making each operator more efficient in his role and allowing the organization to benefit from the contribution that human beings bring to the production processes [7].

Industry 5.0 is structured around five core dimensions: collaboration between humans and machines, coordination of production processes across systems and departments, communication between operators and automated components, automation of repetitive and hazardous tasks and data analytics to support real-time decision making. Importantly, these dimensions are not

oriented toward throughput maximization alone, but toward resilient value creation and human well-being. This distinguishes Industry 5.0 from its predecessor: where Industry 4.0 followed efficiency as a primary objective, Industry 5.0 frames productivity as one goal among several, to be pursued in balance with the needs of the human workforce [9].
Building on the concepts introduced above, Industry 5.0 is built on three pillars [7]:

- **Human-centricity**: technology is designed to follow and serve human needs and capabilities. Workers are empowered rather than replaced. This implies a fundamental redesign of the human-machine interface: instead of forcing operators to navigate complex dashboards and rigid workflows, systems should adapt to the natural communication patterns and to the expectations of the people who use them.

- **Sustainability**: production processes are designed to minimize environmental impact. Industry 5.0 explicitly frames manufacturing as an actor with environmental responsibilities, not only economic ones.

- **Resilience**: industrial systems are designed to be adaptable and robust to face technological, environmental or social disruptions. The COVID-19 pandemic, which exposed critical fragilities in global supply chains, was one of the motivating factors behind the Commission's attention on this pillar [7].

The human-centric pillar has direct implications for the design of human-machine interfaces in manufacturing environments.

Traditional MES interfaces are typically based on predefined forms, fixed dashboards and structured query languages. While effective for standardized and repetitive tasks, these interfaces present significant limitations when users need to express complex or non-standard information needs. Accessing production data through conventional interfaces requires either technical knowledge of the underlying database schema and query language, or the availability of pre-built reports that exactly matches the requested information. In practice, this means that a large part of the informational content stored in the MES is effectively inaccessible to the majority of operators, who lack the technical background to formulate structured requests. As a result, information requests are frequently delegated to technical staff or IT departments, introducing delays that can be critical in a production environment where decisions must be made in real time.

This limitation has direct operational consequences. When an operator on the shop floor needs to know the status of a specific production order, the remaining materials for a given batch, or the cause of a recent machine downtime, the inability to obtain this information quickly and autonomously forces him to rely on intermediaries or to wait for scheduled reports. This slows down the decision-making process and also limits the ability to react proactively to production events. In a manufacturing context where reactivity and flexibility are increasingly central competitive factors, this informational friction represents a structural inefficiency.
Natural language interfaces powered by Large Language Models address this problem directly. By enabling any operator to query the MES in his own language, without requiring knowledge of SQL, database schemas or dedicated software tools, these systems liberalize the access to production information. Each query can be resolved in seconds, without technical intermediaries and without relying on pre-built reports. Rather than requiring each user category to navigate dedicated GUI pages or undergo specialized training, a natural language interface can serve all these roles simultaneously, adapting its responses to the specificity of the question [10].

In this sense, conversational interfaces, based on LLMs are not only a technological trend: they represent a concrete implementation of the human-centric vision articulated by Industry 5.0, bringing the informational richness and complexity of a modern MES to operators on the shop floor, regardless of their technical background. Furthermore, by facilitating both horizontal communication, among operators who share information about ongoing production activities and vertical communication, between the shop floor and management levels, these interfaces contribute to a more connected, informed and reactive organization, fully aligned with the collaborative principles at the core of Industry 5.0 [9].

2.2 Manufacturing Execution Systems

Manufacturing Execution Systems (MES) are information systems used in manufacturing, designed to monitor and optimize the transformation of raw materials into finished goods. MES provides real time information about multiple elements of the production processes. The system is in communication between the ERP systems (enterprise resource planning) and production machines, and it manages and integrates heterogeneous data related to production processes, such as machine states, production orders, quality metrics and operational events. In practice, MES platforms serve as central reposi-

tories of manufacturing knowledge [11].

The Manufacturing Enterprise Solutions Association International (MESA) defined eleven core functions of Manufacturing Execution Systems, known as the MESA-11 model. Although the model has evolved over time, these functions still represent a baseline for modern MES platforms [12]:

- Resource allocation and status
- Operations/detailed scheduling
- Dispatching production units
- Document control
- Data collection and acquisition
- labour management
- Quality management
- Process management
- Maintenance management
- Product tracking and genealogy
- Performance analysis

From an operational perspective, MES is the digital structure of the shop floor, continuously collecting, organizing and linking data generated during production. At any given time, the system returns an updated representation of the manufacturing state, including which orders are being processed, on which machines, by which operators and under which process parameters.
In practice, MES platforms support a wide range of daily manufacturing activities. These systems can track a large amount of data, maximizing production efficiency and cost savings.
Other advantages related to the implementation of a MES are: [12]

- **Quality control improvement**: Events such as machine breakdowns, deviations from process parameters or nonconformances are captured in real time and associated with the corresponding production context. Problems are identified immediately. In this way, manufacturing company are able to reduce waste, scraps and reworks.

- **Increased uptime**: MES generates realistic production schedules by balancing personnel, material and equipment resources, maximizing productive flow and asset utilization.
- **Reduced inventory**: MES updates inventory with new production materials, scrap and nonconformances, so purchasing, shipping and scheduling departments know exactly what materials are available. This allows a better management of the inventories and a cost reduction.
- **Paperless production**: Eliminating paperwork reduces the possibility of human error and data recorded from the production floor is immediately available across all integrated systems.
- **Improved product traceability**: MES tracks the entire production cycle from start to finish, grouping final parts or batches with their corresponding production data.

Furthermore, MES platforms often enforce standardized operating procedures by guiding operators through digital work instructions and preventing the execution of non authorized operations and so preventing errors and waste of resources and money.

Human interaction with a MES typically occurs through dedicated user interfaces tailored to different roles. Operators interact with the system to start and complete operations, report production quantities, acknowledge alarms or record quality inspections. Production supervisors and engineers use MES dashboards to monitor key performance indicators, analyze bottlenecks, investigate problems and adjust production schedules. At a higher level, managers rely on aggregated MES data for performance analysis and decision making.
Despite this central role, traditional MES interfaces are usually based on predefined forms and fixed dashboards. While these interfaces can be effective for standardized tasks, they limit flexibility and make it difficult for non technical users to express complex information needs, making data analysis time consuming. This limitation motivates the exploration of more natural and adaptive interaction mechanisms, such as conversational interfaces based on Large Language Models.
In this context, Large language Models and Retrieval Augmented Generation (RAG) emerges integrating AI Models with external knowledge sources (MES databases and document repositories). RAG based systems combine a retrieval component, which selects relevant information from a knowledge

base, with a generative model that conditions its outputs on the retrieved content. This allows language models to use domain specific and updated data when generating responses, instead of relying exclusively on their internal parameters.
From the perspective of reducing the labor cost of software development and maintenance, emerging software development techniques have explored paradigms such as no code and low code development platforms. Recent research has explored the use of large language models for interacting with MES, giving rise to so called Chat-with-MES systems [1].
The integration of MES data with LLMs enables the use of language models as flexible interfaces for querying and manipulating data over manufacturing information, allowing users to express information needs in an intuitive and human centric way. Efficient and friendly interaction will significantly reduce the workload of workers and the training cost for new employees [13].

The integration of MES data with conversational LLM-based architectures therefore enables flexible and reliable access to manufacturing information, supporting both information retrieval and higher-level reasoning tasks. By abstracting the underlying system complexity, Chat-with-MES approaches allow users to express information needs in natural language, significantly improving usability and decision support in industrial environments. A more detailed discussion of Chat-with-MES and its application is provided in the following chapter.

2.3 Artificial Intelligence and Large Language Models

The development of Artificial Intelligence and, in particular, of Large Language Models represents the technological foundation that makes natural language interfaces for MES systems possible.

Natural Language Processing (NLP) is a subfield of artificial intelligence concerned with enabling machines to understand, interpret and generate human language. Natural language processing aims to formulate algorithms and techniques that empower computers to comprehend and interpret human language through natural language understanding (NLU) and generate text through natural language generation (NLG).
Early approaches to NLP relied on rule based systems. Even though these methods were conceptually solid, they were difficult to scale, especially when

exposed to the variability and complexity of real world language. The advent of machine learning marked a paradigm shift, moving NLP from explicit linguistic rules toward data driven statistical modeling, in which linguistic patterns were learned from data instead of being explicitly encoded through every possible rule.
A language model aims to assign a probability to a sequence of words. Given a sentence of length T,

$$w_1, w_2, \ldots, w_T,$$

the joint probability of the sequence can be expressed using the chain rule of probability as

$$P(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_1, w_2, \ldots, w_{t-1}).$$

This formulation shows that language modeling reduces to predicting the probability of the next word given all preceding words. However, directly estimating the conditional probability

$$P(w_t \mid w_1, \ldots, w_{t-1})$$

becomes computationally expensive for large vocabularies and long contexts and is severely affected by data sparsity. To make probability estimation tractable, traditional statistical language models introduce the Markov assumption, which limits the conditioning context to a fixed number of preceding words.
An n-gram model is a language model that approximates the conditional probability of a word by considering only the previous $n-1$ tokens [2], where n is a fixed model order and typically $n \ll T$:

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-(n-1)}, \ldots, w_{t-1}).$$

Under this assumption, the probability of a word depends only on a local context of fixed length. For example, a unigram model ($n = 1$) assumes complete independence between words,

$$P(w_t),$$

[2]Tokens are obtained by segmenting raw text into smaller units, which may correspond to entire words, subwords, individual characters, or punctuation symbols, depending on the tokenization strategy employed. Modern Large Language Models typically rely on subword-based tokenization methods

while a bi-gram model ($n = 2$) conditions each word on the immediately preceding token,

$$P(w_t \mid w_{t-1}).$$

While unigram and higher-order n-gram models are computationally efficient and conceptually simple, they are limited in their ability to capture longer dependencies and semantic structure in natural language. These limitations motivate the transition toward neural language models and, more recently, large scale Transformer based architectures [14].

Neural Language Models (NLMs) were developed to go beyond the limitations of n-gram models, which could only consider a fixed number of preceding words and struggled with sequences they had never seen before. Rather than estimating probabilities through discrete counts, NLMs represent words and contexts as continuous vectors, allowing the model to capture semantic similarities and generalize across unseen combinations.
As in statistical language modeling, the objective remains to estimate the conditional probability of the next word in a sequence. However, instead of applying the Markov assumption, neural language models map the entire preceding context into a continuous vector representation, which is then used to predict the next token.
A key innovation of neural language models is the use of word embeddings. Each word in the vocabulary is represented as a vector, allowing the model to capture semantic and syntactic similarities between words. This continuous representation decreases the data sparsity problem present in n-gram models [14].

Despite the introduction of continuous representations through word embeddings, early neural language models relied on recurrent architectures. Recurrent neural networks process sequential data by maintaining a hidden state that is updated at each time step and encodes information from previous inputs [15]. However, these architectures showed significant limitations in modeling long-range dependencies and in scaling efficiently to large datasets. In particular, recurrent models process tokens sequentially, resulting in limited parallelization. This sequential nature makes training extremely slow and computationally expensive, requiring significantly more time and hardware resources compared to architectures that can process all tokens simultaneously, which in practice limited the scale at which these models could be trained.
The Transformer architecture was introduced to try to solve these limitations relying on the "self attention" mechanism. Self attention allows each

token in a sequence to compute a weighted representation of all other tokens, where the weights reflect their relevance for interpreting the current token within the given context, enabling the model to explicitly capture global dependencies regardless of their distance in the sequence. These weights are "learned" dynamically and reflect the relative importance of different tokens when predicting a given word [16].
A key advantage of the Transformer architecture is its high degree of parallelism. Since all tokens are processed simultaneously, transformers can be trained efficiently, making them particularly suitable for large scale learning. As a result, Transformers have become the dominant architecture for language modeling and form the foundation of modern Large Language Models [14].

Large Language Models (LLMs) are neural language models based on the Transformer architecture and trained on massive amount of text. From a probabilistic perspective, LLMs share the same fundamental objective as earlier language models, estimating the conditional probability of the next token given the preceding context, but achieve this through attention-based architectures and a significantly larger number of parameters.
The defining characteristic of LLMs is scale. By increasing model capacity, training data size and computational resources, these models show improved performance across a wide range of language tasks. Scaling can lead to emergent behaviors, including in context learning and improved generalization to tasks not explicitly seen during training.
LLMs are typically pre-trained using a self-supervised objective, where the model learns to predict the next token in a sequence, given its preceding context. This pre training phase allows the model to acquire a broader linguistic and semantic knowledge from large scale textual data.
This general knowledge can then be adapted to specific tasks either through fine-tuning, where the model is further trained on a smaller dataset, or through prompting, where the desired behavior is described directly in the input text without modifying the model parameters. This means that a single pre-trained model can be reused across many different applications, avoiding the need to train a new model each time. This significantly reduces the time, data, and computational resources required to deploy AI systems in new contexts [14].

State of the art

3.1 Text-to-SQL and MES

In recent years, the emergence of Large Language Models has enabled new human–MES interaction paradigms in which users can access, explore and manipulate manufacturing data through natural language interfaces. This chapter reviews the current state of the art in LLM-based interaction mechanisms for Manufacturing Execution Systems, to enhance usability, flexibility and decision support in industrial environments.

Beyond the specific Text-to-SQL problem, prior work has also examined the broader question of how users interact with MES interfaces. A well-organized management interface is essential for any MES [1] and that the graphical user interface (GUI) plays a central role in conveying information and supporting decision-making on the shop floor. Considering that manufacturing environments differ significantly from one factory to another, developing and customizing these interfaces is costly; no-code/low-code platforms cut interface development costs but lose extensibility for complex manufacturing scenarios.
Traditional MES interfaces are also hard to learn. Combining LLMs/LLM-agents with RPA (Robotic Process Automation) is starting to make systems more human-centered and chatbot assistants for shop-floor coordination have been proposed.
Considering companies' growing desire to extract more information from data and the increasing volume of available data, the need to query information from production processes has also increased. Querying databases often requires to know the structure of the database and the use of SQL which can represent a technical barrier for non technical users. To overcome this limitation, Large Language Models have been introduced. LLMs simplify the

task of translating natural language queries into precise SQL statements.

Text-to-SQL is the task of automatically translating a natural language question into an executable SQL query against a relational database. It represents one of the most practically significant applications of natural language processing, as it enables non-technical users to access structured data without requiring knowledge of SQL or the underlying database schema.

The task can be formally defined as follows: given a natural language question q and a database schema $\mathcal{S}$, consisting of a set of tables, columns, data types and foreign key relationships, a Text-to-SQL system must produce a SQL query s such that executing s against the database returns the correct answer to q. The correctness of the generated query is typically evaluated using execution accuracy, defined as the fraction of questions for which the execution result of the generated SQL matches the execution result of the ground truth SQL:

$$A = \frac{|\{q \in \mathcal{Q} : \text{exec}(s_q) = \text{exec}(g_q)\}|}{|\mathcal{Q}|} \tag{3.1}$$

where $\mathcal{Q}$ is the set of benchmark questions, s_q is the generated SQL for question q and g_q is the corresponding ground truth SQL (this approach will be introduced in practice in Chapter 4 and 5).

A text-to-SQL system enables the user to extract specific information without the need to explicitly write the SQL query but interacting with natural language, making complex data more accessible [17].
However, building reliable text-to-SQL systems is highly challenging. The complexity arises from several factors:

- **Ambiguity and Linguistic Complexity**: Natural language queries may contain ambiguous or unclear expressions that cannot always be correctly interpreted by the LLM. The use of inaccurate terms can lead to incomplete information, which may result in the generation of incorrect queries for the information system.

- **Database schema Understanding**: Text-to-SQL systems must understand the structure of the database in order to generate correct SQL queries. Database schemas are often large and complex and they can vary significantly through different domains. This variability makes it difficult to correctly associate natural language inputs with the corresponding database elements.

- **Rare and Complex SQL Operations**: Some queries require advanced SQL operations such as sub-queries, joins between tables, or aggregations. Models may struggle to generate these operations correctly.
- **Cross-Domain Generalization**: Text-to-SQL models trained on a specific domain may not generalize well to other domains. Differences in schema structures, terminology and query patterns can significantly impact model performance across different databases.

These challenges make the development of robust and generalizable text-to-SQL systems an active research area in the field of natural language interfaces for databases [17] [18].

The implementation of LLM-based text-to-SQL systems can generally be divided into three main components [18]:

- **Question understanding**: the generated SQL query must accurately reflect all the relevant information expressed in the user's natural language input.
- **Schema comprehension**: the database schema defines the structure of tables and columns. The text-to-SQL system must identify the relevant tables and attributes that correspond to the user's question.
- **SQL generation**: after understanding the query and the schema, the model must generate the correct SQL syntax to produce an executable query capable of retrieving the desired information.

LLM-based text-to-SQL systems have shown promising results, but they still present several limitations. In particular, they often struggle to correctly interpret complex database schemas, generate SQL queries for complicated requests and generalize effectively across different application domains.

An approach that is capable of overcoming these limitations is Retrieval Augmented Generation (RAG). The main objective of RAG is to enhance the capabilities of Large Language Models by retrieving relevant information from external sources, such as databases or documents, before generating a response. This allows the model to rely not only on the knowledge encoded in its internal parameters but also on external and potentially updated information. When applied to Text-to-SQL tasks, the system can support the query process in different ways. Retrieved context can help the model to generate

more accurate SQL queries by providing relevant schema details or examples of similar queries. In other situations, the system may directly retrieve the requested information from available data sources, reducing or eliminating the need to generate an explicit query. Both modalities significantly improve data accessibility, lowering the technical barrier for users who need to interact with database information without possessing specialized knowledge of SQL or the underlying schema.

By integrating external knowledge into the generation process, RAG-based approaches can improve the interpretation of database schemes, reducing ambiguities in natural language queries and increasing the adaptability of text-to-SQL systems without requiring additional model training. In the context of text-to-SQL systems, the retrieved information may include database schema descriptions, examples of SQL queries or domain-specific documentation.
In the article *"Chat with MES: LLM-driven user interface for manipulating garment manufacturing system through natural language"*, it is shown an example of "LLM-based Text2SQL", in which natural language is converted in executable SQL queries. However, according to a large-scale Text-to-SQL benchmark, the most effective models achieve an execution accuracy of 54.89%, whereas human experts achieve approximately 92.96% [19], highlighting a significant gap that remains between automated systems and human performance in the general case. Nevertheless, this gap can be substantially reduced in domain-specific applications: by incorporating additional knowledge through prompt engineering, the performance of Text-to-SQL systems in practical deployments can be improved significantly.

In real-world applications, effectiveness depends on the integration of domain-specific business knowledge, which helps to interpret correctly user requirements and produce executable commands. Recent approaches, such as "ChatBI" [20], extend this paradigm by enabling Natural Language interaction with Business Intelligence systems, especially in scenarios with large numbers of columns. Other solutions, including "ChatDB" [21] and "DinSQL" [22], enhance LLM capabilities by supporting multi-step and dynamic SQL generation for more complex database operations. However, most existing LLM-based systems are designed only for query tasks that are not able to modify data, in order to avoid risks related to incorrect data modifications. In contrast to this, MES environments require precise execution of update, insert and delete operations, where both the selection and the sequence of actions must be fully reliable. This introduces strict requirements for the agent, which must understand database schemas, relationships and production con-

straints.

As an extension of Text-to-SQL, ChatBI [20] was proposed to address the specific challenges of Natural Language to Business Intelligence query translation, particularly in scenarios involving a large number of columns and complex analytical queries. Traditional Text-to-SQL systems struggle when the database schema contains hundreds of columns, as the model must identify the relevant subset without explicit guidance. ChatBI addresses this by introducing a column selection mechanism that filters the schema before SQL generation, reducing the input complexity and improving the accuracy of the generated queries. This approach is particularly relevant for enterprise data warehouses and BI systems, where schemas are typically wide and heterogeneous.

ChatDB [21] takes a different approach by augmenting LLMs with a database using as symbolic memory. Rather than generating a single SQL query to answer a question, ChatDB decomposes complex requests into a sequence of simpler SQL operations, executes them step by step and stores intermediate results in the database for subsequent steps to use. This "chain-of-thought" approach to database interaction allows the system to handle multi-step queries that would be difficult or impossible to express as a single SQL statement. Each intermediate result is stored and referenced by name in subsequent steps, enabling the model to reason over structured data incrementally rather than generating monolithic queries.

DinSQL [22] addresses the complexity of SQL generation through decomposed in-context learning with self-correction. The system breaks the Text-to-SQL task into a pipeline of subtasks: schema linking, query classification, SQL generation and self-correction. In the schema linking step, the model identifies the tables and columns relevant to the question. In the classification step, the query is categorized by complexity, easy, medium, or nested, before being routed to a specialized sub-prompt. The self-correction step then reviews the generated SQL for syntactic and semantic errors and attempts to fix them before execution. This modular approach achieved state-of-the-art results at the time of its publication, demonstrating that decomposing the generation task into smaller, more manageable subtasks significantly improves accuracy on complex queries.
These three approaches, ChatBI, ChatDB and DinSQL, represent complementary strategies for improving Text-to-SQL performance: column-level schema filtering, multi-step symbolic reasoning and task decomposition with self-correction respectively. However, all three are designed primarily for

read-only query scenarios. However, MES environments require precise execution of write operations.
The Chat with MES (CWM) framework [1] was proposed to address precisely these requirements, extending the Text-to-SQL paradigm to the full complexity of MES environments. Rather than relying on a single prompt to generate a SQL query, CWM organizes the interaction into a structured pipeline composed of three specialized components. The first component, Query Rewriting, pre-processes the user's natural language input by resolving ambiguities and aligning the terminology of the question with the domain vocabulary of the MES database. The second component, Multi-Step Operations, handles requests that require sequences of database operations, including write operations such as INSERT, UPDATE and DELETE, by decomposing them into ordered, verifiable steps that can be executed safely and reliably. The third component, Text-to-SQL generation, translates the preprocessed query into an executable SQL statement, guided by explicit schema information and domain-specific rules injected into the prompt.

Building on the CWM architecture, a further contribution of modern LLMs is scalability. By dynamically injecting the database schema at runtime, retrieved directly from the target database rather than hard-coded into the system, a single prompt can be designed to operate across different MES installations without requiring manual adaptation. This means that the same prompt structure, combined with an automated schema scanning mechanism, can be applied to any relational database regardless of its specific tables, columns and relationships. The LLM reads the schema, understands the structure and generates the appropriate SQL query. What previously required domain-specific fine-tuning or manual rule engineering can now be achieved through a well-designed prompt, significantly reducing both the development effort and the technical barrier for deploying Text-to-SQL systems in new industrial contexts. In the present work, this mechanism is implemented through the "build_schema_markdown()" function, which queries the database at runtime to extract the current schema and inject it directly into the prompt, as described in detail in Chapter 5.

This modular architecture achieved 86% execution accuracy on a garment MES database using GPT-4o as the underlying model, as reported by the authors [1]. The present work builds on this result. Starting from an independently measured baseline of approximately 60%, a single optimized prompt incorporating explicit domain rules and structured reasoning was developed and evaluated across three state-of-the-art LLMs, achieving approximately 86% execution accuracy as measured by the evaluation framework described

in Chapters 5 and 6.

Methodology

4.1 Proposed Framework

The experimental setting considered in this work is based on "Chat with MES" framework [1], a system designed to allow users to interact with a Manufacturing Execution System through natural language. The idea is to adopt this database and to improve the performance of the Text-to-SQL component by analyzing the failure cases of the original system and introducing a series of targeted adaptations to the prompt. The original implementation, in fact, was designed and optimized for a single model and struggled to produce correct SQL queries for more complex questions involving multi-table joins and ambiguous natural language requests. The goal of this work is to bring the execution accuracy to a significantly higher level through iterative prompt engineering; on the other hand, to evaluate whether the optimized prompt retains its effectiveness across three fundamentally different model architectures, providing a measure of the robustness of the model.
As anticipated, the underlying database models a clothing production environment and includes several interconnected tables covering the full lifecycle of a production order: customers and their orders, product types and product variants with their attributes, raw materials and work in progress (WIP) items, warehouse stock and material allocations, cutting and sewing tasks assigned to working groups and the progress tracking of each production stage. This rich relational structure, with its specific terminology and complex table joins, makes it a challenging and realistic benchmark for Text-to-SQL evaluation.
The three Large Language Models selected for this benchmark are GPT-5 (OpenAI), Claude Opus 4 (Anthropic) and Gemini 2.5 Pro (Google). These models represent the state of the art offerings from the three leading AI research organizations at the time of writing and were selected to ensure a

representative comparison.
The system is structured as a pipeline composed of five main files. Three of them handle the core processing flow, transforming a natural language question into a natural language answer through SQL generation, database execution and response synthesis. The remaining two are dedicated to the evaluation infrastructure: one manages the automated execution of multiple evaluation cycles across the selected models, while the other compares the generated SQL results against the ground truth and computes the execution accuracy. The primary objective of this implementation is not simply to build a functional Text-to-SQL interface, but to rigorously and objectively evaluate its reliability and accuracy across the three selected models in a statistically robust way.
To achieve this, the pipeline is designed to support repeated, automated evaluation cycles over a fixed set of 55 questions, each paired with a ground truth SQL query previously manually verified against the database. For each model, 10 independent cycles were executed, resulting in 550 query evaluations per model and 1,650 total evaluations across the three models. Running multiple independent cycles and aggregating the results statistically minimizes the impact of the non deterministic nature of LLM outputs, since the same prompt may produce different SQL queries across different calls.

The prompt engineering process was carried out iteratively using Claude Opus 4 as the reference model, running repeated evaluation cycles and analyzing the failure cases after each iteration to identify and fix structural mismatches between the generated SQL and the ground truth. Once the prompt reached an execution accuracy of approximately 86%, the evaluation was extended to GPT-5 and Gemini 2.5 Pro, without any modification to the prompt, to verify how robust the design is when applied to fundamentally different model architectures. In other words, whether a prompt developed and optimized for one model continues to work well when applied to a completely different one without any modification.

These five files have different roles and the figure below (4.1) represents the call flow between the main functions of the files, showing how each component invokes the others to transform a natural language question into a verified answer.

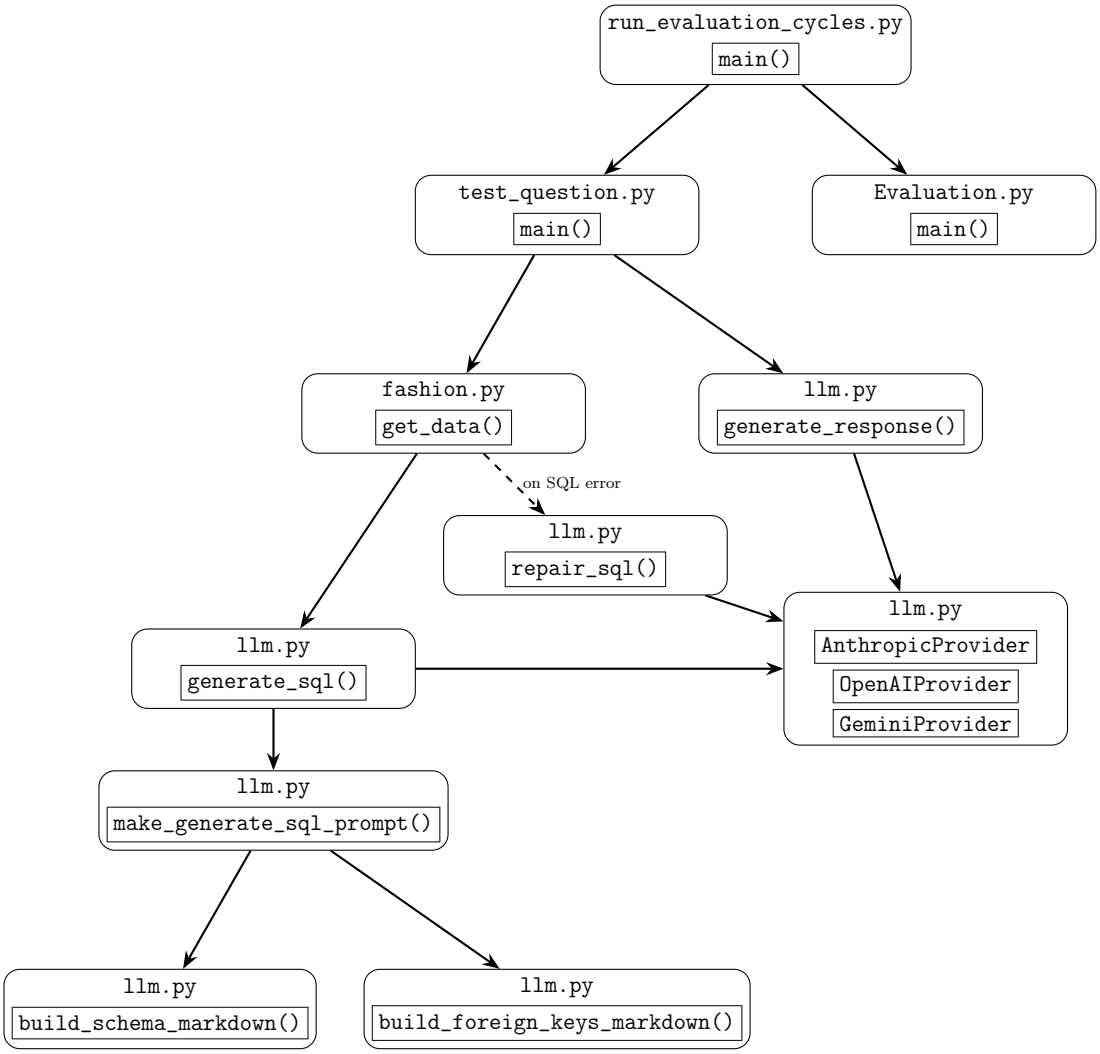


Figure 4.1: Architecture of the Text-to-SQL system pipeline

A further significant improvement introduced in this work concerns the representation of the database schema within the prompt. In the original implementation, the schema was provided as a static string hard coded directly into the prompt. Table names, column names, data types and foreign key relationships were written manually and remained fixed regardless of the actual state of the database. This approach, while functional for a stable schema, presents a critical limitation in a real manufacturing environment: any modification to the database structure, such as the addition of a new table, the renaming of a column or the introduction of a new foreign key constraint, would require a manual update of the prompt, with the risk of inconsistencies

between the prompt and the actual schema going undetected.

In the new implementation, the schema representation is generated dynamically at runtime by querying the MySQL information_schema directly. The functions build_schema_markdown() and build_foreign_keys_markdown() connect to the database at each invocation and retrieve the current table structure and foreign key relationships, producing a representation that is always consistent with the actual state of the database. This makes the system adaptive: if a new table is added to the MES database or an existing one is modified, the prompt is automatically updated at the next execution without any manual intervention. In a production environment where the database schema evolves over time as new production processes or product categories are introduced, this dynamic approach represents a substantial practical advantage over a static prompt, ensuring that the Text-to-SQL system remains accurate, without requiring continuous maintenance of the prompt itself.

At a macro level, the system performs five main activities to transform a natural language question into a verified answer.

Stage 1 — Orchestration.

The entry point of the system is run_evaluation_cycles.py, which launches and controls the entire benchmark. For each of the $N = 10$ evaluation cycles, it sequentially invokes test_question.py and Evaluation.py as subprocesses, passing the selected LLM model name as input. After each cycle, it collects the execution accuracy and the list of mismatched question IDs, updating a frequency counter that tracks how often each question fails across cycles. At the end of all cycles, the aggregate statistics are computed and saved to cycles_report.json.

Stage 2 — Question processing.

For each of the 55 input questions, test_question.py reads the natural language query from the corresponding JSON file and retrieves data thanks to fashion.py by calling get_data(). This function is responsible for generating the SQL query, executing it against the MySQL database and returning the result rows. Once the rows are available, test_question.py calls generate_response() from llm.py to translate the SQL output into a natural language answer. Both the generated SQL and the response are saved to a JSON output file in the Output/ directory.

Stage 3 — SQL generation and execution.

Inside fashion.py, get_data() calls generate_sql() from llm.py, which builds a schema-aware prompt by injecting the full database structure into the context and submits it to the selected LLM through the appropriate provider class. The model returns a JSON object containing a reasoning trace and the generated SQL query. The query is then validated, only read-only SELECT and WITH statements are accepted and executed against the MySQL database using pymysql. If the execution raises a database exception, fashion.py automatically invokes repair_sql() from llm.py, passing the original query, the failed SQL and the error message. The repaired SQL is validated and executed again. The final result, including the SQL, the rows, the execution time, the token counts and a repaired flag, is returned to test_question.py.

Stage 4 — Prompt construction.

Every call to generate_sql() and repair_sql() relies on a function, make_generate_sql_prompt(), that dynamically builds the system prompt at runtime by querying the MySQL information_schema to extract the current table structure and foreign key relationships.
build_schema_markdown() produces a representation of all tables and their columns, including data types, nullability and key constraints.
build_foreign_keys_markdown() produces a list of all foreign key relationships in the form table.column → referenced_table.referenced_column.

These two representations are injected into the prompt together with a domain glossary containing explicit rules for status values, table disambiguation, column selection, aggregation patterns and response format, ensuring that the model has all the context needed to produce a correct and executable SQL query.

Stage 5 — Evaluation.

After all 55 output files have been written, Evaluation.py loads each output file alongside the corresponding input file, extracts the generated SQL and the ground truth SQL and executes both against the database. The two result sets are compared using a value based matching strategy that ignores column names and aliases, comparing only the actual values contained in each row. Questions corresponding to write operations, INSERT, UPDATE, DELETE, are automatically skipped since they carry no evaluable ground truth SQL,

reducing the comparable set from 55 to 44 questions per cycle. The outcome of each comparison is aggregated into an execution accuracy score and saved to evaluation_results.json, which is then read by run_evaluation_cycles.py to update the cycle statistics.
Having outlined the end-to-end flow of the pipeline, the chapter 5 describes each file in detail, explaining the role of every function. The files are presented in top-down order, following the hierarchical structure of the system from the orchestration layer to the LLM abstraction and the evaluation module.

4.2 Evaluation Methodology

The evaluation of a Text-to-SQL system requires a metric that is both objective and reproducible. Two fundamentally different approaches exist in the literature: result set comparison and LLM as judge evaluation.
The original Chat with MES framework [1] adopts an LLM-as-judge approach, in which a separate GPT-4 instance receives the original question, the ground truth answer and the generated answer and produces a binary correctness judgment. While this approach can capture semantic equivalence between answers expressed in different forms, it introduces a source of non-determinism that makes the evaluation metric itself a random variable.

Formally, let $J(q, a_{gt}, a_{pred})$ be the judgment function of the LLM judge for question q, ground truth answer a_{gt} and predicted answer a_{pred}. Since LLMs are stochastic by nature, J is not a deterministic function, running the same evaluation twice may produce different outcomes [1], so that:

$$P\left(J(q, a_{gt}, a_{pred}) = 1\right) \neq 1 \quad \text{even when } a_{pred} \equiv a_{gt} \tag{4.1}$$

This non-determinism propagates directly into the accuracy estimate. If the judge produces different verdicts across runs, the measured accuracy $\hat{A}$ becomes a random variable rather than a fixed quantity, making it less reliable to compare results across different runs or different systems without knowing the variance of the judge itself. Additionally, using an LLM as an evaluator introduces a potential bias: the judge may favor or penalize certain styles of answer, certain phrasings, independently of whether the content is correct.

For these reasons, this work adopts a deterministic, result set-based evaluation metric: execution accuracy.

Given a question q_i with ground truth SQL g_i and generated SQL s_i, both queries are executed against the database and their result sets $R(g_i)$ and $R(s_i)$ are compared directly. Each result set is a table of rows, where each row represents one record returned by the query, for example, one order, one task, or one material. The number of rows therefore reflects the number of entities that satisfy the conditions of the query: if the ground truth returns 2 orders and the generated SQL returns 3, the extra row indicates that the model applied incorrect filters or performed incorrect joins and the result must be considered wrong regardless of whether the individual values are correct. For this reason, the first condition of the match function checks that the two result sets have the same number of rows. Only when the row counts are equal does the comparison proceed to the value level: each row is converted into a set of normalized values, discarding column names and aliases and the ground truth value set must be a subset of the predicted value set for each paired row:

$$\text{match}(q_i) = \begin{cases} 1 & \text{if } |R(g_i)| = |R(s_i)| \text{ and } \forall j : \text{vals}(R(g_i)_j) \subseteq \text{vals}(R(s_i)_j) \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

where $\text{vals}(r)$ denotes the set of normalized values in row r. The execution accuracy over the set of comparable questions $\mathcal{Q}$ is then:

$$A = \frac{\sum_{i \in \mathcal{Q}} \text{match}(q_i)}{|\mathcal{Q}|} \tag{4.3}$$

This metric is fully deterministic: the same generated SQL always produces the same evaluation outcome, regardless of how many times the evaluation is run. This property is essential for the statistical analysis performed in this work, where the mean and standard deviation of accuracy are computed across 10 independent generation cycles.

4.3 Prompt Engineering

The task of adapting a Large Language Model to a specific domain can be approached in fundamentally different ways [23]. The most common distinction in the literature is between fine-tuning and in-context learning. Fine-tuning consists of updating the weights of a pre-trained model on a domain-specific dataset of question-SQL pairs, so that the model internalizes the patterns

of the target domain through gradient descent. In-context learning, by contrast, does not modify the model weights at all, instead, it encodes the domain knowledge directly in the prompt, guiding the model's behavior through carefully structured instructions, examples and constraints.
This work adopts prompt engineering as its primary methodological approach, for three main reasons.

The first reason is practical accessibility. Fine-tuning a large language model requires a labeled dataset of sufficient size, typically hundreds or thousands of annotated question-SQL pairs, as well as significant computational resources for training. In a manufacturing context, such datasets are rarely available: MES databases are highly customized, the queries that operators need to perform are often implicit and undocumented and the cost of annotation is prohibitive.

Prompt engineering, by contrast, requires no training data and no computational resources beyond the API calls needed for evaluation. A well-engineered prompt can be deployed immediately on any database without any preparatory phase.

The second reason is portability. A fine-tuned model is tied to the specific database schema and question distribution on which it was trained. Any change to the schema, such as the addition of a new table or the renaming of a column, may degrade its performance without retraining. A prompt-based system, especially one that injects the schema dynamically at runtime as done in this work, adapts automatically to any structural change in the database. This property is particularly valuable in manufacturing environments, where MES databases evolve continuously as new production processes or product categories are introduced.

The third reason is model independence. A fine-tuned model is architecture-specific: a model fine-tuned on GPT-4 cannot be directly applied to Claude or Gemini. A prompt, by contrast, can be applied to any model that understands the same language. This property is central to one of the research questions of this work: whether a prompt optimized for one model architecture retains its effectiveness when applied to fundamentally different architectures. Verifying this property requires a model-agnostic approach, which fine-tuning by definition cannot provide.

The prompt engineering process adopted in this work follows an iterative refinement methodology driven by systematic failure analysis. Starting from

the baseline prompt of the Chat with MES framework, the prompt was progressively enriched with domain-specific rules derived from the inspection of ground truth SQL queries for each failing question. Each iteration of the prompt was evaluated over ten execution cycles and the mismatch frequency of each question was tracked across cycles to distinguish structural failures, caused by missing or incorrect rules in the prompt, from stochastic failures caused by the non-deterministic nature of LLM generation.
It is important to note that the prompt engineering process was conducted exclusively using one model, Claude Opus 4, as the reference model. The prompt was then applied without modification to GPT-5 and Gemini 2.5 Pro, to assess whether the improvements achieved through prompt engineering on one architecture generalize to other architectures. This experimental design separates the contribution of the prompt from the contribution of the model and allows drawing conclusions about the robustness of the prompt design itself.

The prompt used for SQL generation evolved through an iterative refinement process driven by the systematic analysis of failure cases observed in each evaluation cycle. The process consisted of three main steps, each introducing a targeted set of domain-specific rules to address the most frequent structural mismatches identified in the evaluation output. Throughout the entire process, the evaluation was conducted exclusively using the deterministic result set comparison method described in previous section.

4.3.1 Step 1 — Baseline: dynamic schema, minimal rules (59% Accuracy)

The starting point of the prompt engineering process is the dynamic schema version of the original Chat with MES prompt. Unlike the static baseline of the original paper, where the database schema was hard coded as a fixed string, this version queries the MySQL information_schema at runtime to inject the current table structure and foreign key relationships into the prompt. The prompt content, however, remained minimal: a generic context description, a small set of general SQL rules and four domain rules covering status value semantics, task table disambiguation, customer name normalization and the prohibition of CONCAT in material queries. The full prompt for this step is reported in Listing 4.1.

```
1 # CONTEXT
2 You are a MySQL expert for a clothing production MES.
```

```
The database name is: `fashion`

# DATABASE SCHEMA
{schema_md}   <- injected at runtime from
   information_schema

# FOREIGN KEY RELATIONSHIPS
{fk_md}       <- injected at runtime from
   information_schema

# RULES
- Generate ONLY read-only SQL (SELECT statements).
- Use only tables/columns that exist in the schema above.
- Prefer explicit JOIN conditions using foreign keys.
- Keep SQL in ONE LINE (no line breaks).

# DOMAIN RULES

## STATUS VALUES
- status = 0 -> not yet started
- status = 1 -> started but not yet completed
- status = 2 -> completed

## TASK TABLE DISAMBIGUATION
- "cutting tasks" -> ONLY cutting_tasks table
- "sewing tasks"  -> ONLY sewing_tasks table
- NEVER mix the two tables

## CUSTOMER NAME
- "PolyU" and "Hong Kong Polytechnic University"
  -> customer_name = 'PolyU'

## MATERIALS
- NEVER use CONCAT or GROUP_CONCAT
- Return m.name as a separate column for each row

## WORKING GROUP NAME
- ALWAYS JOIN working_group wg and return wg.name AS
   working_group_name
- NEVER return working_group_id alone

## PRODUCTS IN ORDER
- SELECT o.id AS order_id, pt.description, p.attributes
```

```
- JOIN: orders -> order_product -> products ->
   products_types
- Never return quantity or number columns

# OBJECTIVE
Based on the user question, first reason step by step,
then produce the correct SQL.

# RESPONSE FORMAT
{
  "thought": "step by step reasoning",
  "sql": "SELECT ...;"
}
```

Listing 4.1: Step 1 prompt — dynamic schema, minimal rules

Evaluating this version on the 44 comparable questions of the benchmark produced an execution accuracy of 59.09%, corresponding to 26 correct matches out of 44. The most frequent failure categories at this stage were: wrong column selection in task listing queries, incorrect aggregation patterns in progress percentage queries, use of CONCAT in inventory queries despite the explicit rule and missing or wrong join paths for material and working group queries.

4.3.2 Step 2 — Domain rules for aggregation and structure (72% Accuracy)

The second step introduced a set of targeted rules derived from the inspection of the ground truth SQL for the most consistently failing questions. The main additions with respect to Step 1 were:

- Progress percentage queries must compute the ratio of completed to planned units using

```
ROUND(SUM(st.completed_number) / NULLIF(SUM(st.
   planned_number), 0) * 100, 2)
```

  grouped by order_product id, with NULLIF to prevent division by zero. This fixed Q3, Q51 and Q54, which were returning NULL values due to incorrect aggregation.

- Top product queries must aggregate using SUM(op.number) from order_product, never SUM(st.completed_number) from sewing_tasks, which counts a different quantity. This fixed Q55.

- Raw material inventory queries must return m.name and SUM(wm.left_number) as separate columns, filtered with HAVING SUM(wm.left_number) > 0 and must never use CONCAT or GROUP_CONCAT to format the output. This fixed Q13.
- Cutting tasks before a date must return only three columns — id, end_time and working_group_name — and must apply a status != 0 filter to exclude tasks that have not yet started. This fixed Q11.
- Sewing tasks before a date must include all tasks regardless of their status, with no status filter at all. This fixed Q12, where the model was incorrectly excluding tasks with status = 0.
- Cutting task queries that filter by date must always add AND wg.type = 'cut' to the working group join, to prevent sewing working groups from appearing in the results. This fixed Q16.
- The working groups list query must return name and type only, ordered by type then name and must never include the id column. This fixed Q22.
- Remaining workload queries must filter using WHERE st.planned_number > st.completed_number without any status filter, and return only the working group name. This addressed Q29.
- Sewing task queries for a specific customer must follow the join path customers → orders → order_product → sewing_tasks. This fixed Q28.
- Warehouse comparison queries must use ORDER BY left_number DESC LIMIT 1 to return a single row. This fixed Q23.

The full prompt for this step is reported in Listing 4.2.

```
# CONTEXT
You are a MySQL expert for a clothing production MES.
The database name is: `fashion`

# DATABASE SCHEMA
{schema_md}

# FOREIGN KEY RELATIONSHIPS
{fk_md}

```

```
# RULES
- Generate ONLY read-only SQL (SELECT statements).
- Use only tables/columns that exist in the schema above.
- Prefer explicit JOIN conditions using foreign keys.
- Keep SQL in ONE LINE (no line breaks).

# DOMAIN RULES

## STATUS VALUES
- status = 0 -> not yet started
- status = 1 -> started but not yet completed
- status = 2 -> completed

## TASK TABLE DISAMBIGUATION
- "cutting tasks" -> ONLY cutting_tasks. NEVER UNION with
    sewing_tasks.
- "sewing tasks"  -> ONLY sewing_tasks.  NEVER UNION with
    cutting_tasks.

## SEWING TASKS BEFORE A DATE
- WHERE st.end_time < '[date]' ORDER BY st.id
- NO status filter. Include ALL tasks regardless of
   status.

## CUTTING TASKS BEFORE A DATE
- WHERE ct.end_time < '[date]' AND ct.status != 0
- Only 3 columns: ct.id, ct.end_time, wg.name AS
   working_group_name

## CUTTING TASKS AFTER A DATE
- WHERE ct.start_time > '[date]' AND wg.type = 'cut'
- JOIN working_group wg ON ct.working_group_id = wg.id

## WORKING GROUP NAME
- ALWAYS JOIN working_group wg and return wg.name AS
   working_group_name
- NEVER return working_group_id alone

## WORKING GROUPS LIST
- SELECT name, type FROM working_group ORDER BY type,
   name
- Return name and type only, never id

```

```
## CUSTOMER NAME
- "PolyU" and "Hong Kong Polytechnic University"
  -> customer_name = 'PolyU'

## MATERIALS
- NEVER use CONCAT or GROUP_CONCAT
- Return m.name as a separate column for each row

## RAW MATERIALS INVENTORY
- SELECT m.name, SUM(wm.left_number) AS total_left
  FROM warehouse_material wm JOIN materials m ON wm.
     material_id = m.id
  GROUP BY m.name HAVING SUM(wm.left_number) > 0
- Return m.name and total_left as SEPARATE columns

## PRODUCTS IN ORDER
- SELECT o.id AS order_id, pt.description, p.attributes
- JOIN: orders -> order_product -> products ->
   products_types
- Never return quantity or number columns

## REMAINING WORKLOAD
- WHERE st.planned_number > st.completed_number
- Return ONLY wg.name AS working_group_name, no status
   filter

## PROGRESS PERCENTAGE
- ROUND(SUM(st.completed_number) /
  NULLIF(SUM(st.planned_number), 0) * 100, 2) AS
     progress_percentage
- GROUP BY op.id

## TOP PRODUCTS
- SUM(op.number) AS total_made FROM order_product op
- Never use SUM(st.completed_number)

## SEWING TASKS FOR A CUSTOMER
- JOIN: customers c -> orders o ON c.id = o.user_id
  -> order_product op ON o.id = op.order_id
  -> sewing_tasks st ON op.id = st.order_product_id
- Filter by customer_name = 'PolyU'

## WAREHOUSE WITH MOST/LEAST OF A MATERIAL
```

```
- SELECT w.name, wm.left_number FROM warehouse w
  JOIN warehouse_material wm ON w.id = wm.warehouse_id
  JOIN materials m ON wm.material_id = m.id
  WHERE m.name = '[material]'
  ORDER BY wm.left_number DESC LIMIT 1

# REASONING REQUIREMENT
Before writing SQL, reason through:
1. Which tables are needed?
2. Which joins are required?
3. Which filters apply?
4. What columns should SELECT return?
5. Any domain-specific rules apply?

# RESPONSE FORMAT
{
  "thought": "step by step reasoning",
  "sql": "SELECT ...;"
}
```

Listing 4.2: Step 2 prompt — domain rules for aggregation and structure

Running the evaluation after these additions produced the following result:

Model	Cycles	Mean Accuracy	Std Dev
Claude Opus 4	10	71.36%	±2.53%

Table 4.1: Execution accuracy after Step 2. Mean over 10 independent cycles.

4.3.3 Step 3 — Ground-truth verified fixes (86%)

The third and final step was based on a systematic inspection of the ground truth SQL for each of the remaining failing questions. For each mismatch, the expected SQL pattern was identified directly from the input JSON files and encoded as an explicit rule in the prompt. The main fixes introduced were:

- Q4: Products-in-order queries must return only order_id, description, and attributes. The quantity column must never be included.
- Q8: Materials used in an order must be retrieved via a UNION of cutting_material_allocation and sewing_material_allocation, returning only DISTINCT material names.

- Q9: Progress percentage queries must GROUP BY product variant, not aggregate all variants into a single row.
- Q10: Unfinished products must be identified by joining order_product with cutting_tasks and filtering where completed_number < planned_number.
- Q13: The CONCAT prohibition was made more explicit. The formatted output belongs in the response text, never in the SQL.
- Q18/Q39: Cutting tasks using a specific material must return only the standard task columns, without material_name, unit, allocated_num, or is_allocated.
- Q23: Warehouse comparison queries must use ORDER BY left_number DESC LIMIT 1 to return a single row.
- Q28: The full join path for sewing tasks of a specific customer was made explicit with the exact column list.
- Q29: Remaining workload queries must return only the working group name, with no aggregation and no status filter.
- Q51: The fastest progress query must use op.number as denominator, not st.planned_number.
- Q54: Progress percentage per working group must GROUP BY working group.
- Q55: Top products must use SUM(op.number) with GROUP BY product variant.

The full final prompt is reported in Listing 4.3.

```
# CONTEXT
You are a MySQL expert for a clothing production MES.
The database name is: `fashion`

# DATABASE SCHEMA
{schema_md}

# FOREIGN KEY RELATIONSHIPS
{fk_md}

# DOMAIN GLOSSARY

```

```
## STATUS VALUES
- status = 0 -> not yet started
- status = 1 -> started but not yet completed
- status = 2 -> completed

## TASK TABLE DISAMBIGUATION
- "cutting tasks" -> ONLY cutting_tasks. NEVER UNION with
    sewing_tasks.
- "sewing tasks"  -> ONLY sewing_tasks.  NEVER UNION with
    cutting_tasks.

## STANDARD TASK SELECT COLUMNS
For any task listing query use EXACTLY these columns:
  st.id, st.start_time, st.end_time, st.planned_number,
  st.completed_number, st.status, wg.name AS
     working_group_name
Do NOT add material_name, unit, allocated_num,
   is_allocated.

## SEWING TASKS BEFORE A DATE
- WHERE st.end_time < '[date]' ORDER BY st.id
- NO status filter. Include ALL tasks regardless of
   status.

## CUTTING TASKS BEFORE A DATE
- SELECT ct.id, ct.end_time, wg.name AS
   working_group_name
- WHERE ct.end_time < '[date]' AND ct.status != 0

## CUTTING TASKS AFTER A DATE
- WHERE ct.start_time > '[date]' AND wg.type = 'cut'

## WORKING GROUPS LIST
- SELECT name, type FROM working_group ORDER BY type,
   name
- Return name and type only, never id

## PRODUCTS IN AN ORDER
- SELECT o.id AS order_id, pt.description, p.attributes
- No order_name, no number/quantity column.

## PROGRESS % PER PRODUCT
- ROUND(SUM(st.completed_number) /
```

```
  NULLIF(SUM(st.planned_number), 0) * 100, 2) AS
     progress_percentage
- GROUP BY p.id, p.attributes, pt.description

## MATERIALS USED IN AN ORDER
- SELECT DISTINCT m.name ... cutting_material_allocation
   ...
  UNION
  SELECT DISTINCT m.name ... sewing_material_allocation
     ...
- Return ONLY m.name.

## RAW MATERIALS INVENTORY
- SELECT m.name, SUM(wm.left_number) AS total_left
- HAVING SUM(wm.left_number) > 0
- NEVER use CONCAT or GROUP_CONCAT.

## REMAINING WORKLOAD
- SELECT wg.name AS working_group_name
- WHERE st.planned_number > st.completed_number
- Return ONLY working_group_name.

## TOP PRODUCTS
- SUM(op.number) AS total_made FROM order_product
- Never use SUM(st.completed_number).

## CUSTOMER NAME
- "PolyU" / "Hong Kong Polytechnic University"
  -> customer_name = 'PolyU'

# SQL RULES
- Exactly ONE read-only SELECT query.
- Return valid JSON only, no markdown fences.

# REASONING REQUIREMENT
Always reason through these 6 steps before writing SQL:
  1. Which tables are needed?
  2. Which joins are required?
  3. Which filters apply?
  4. What columns should SELECT return?
  5. Any domain-specific terms?
  6. Is this a listing, aggregation, or per-group query?

```

```
# RESPONSE FORMAT
{
  "thought": "step-by-step reasoning covering all 6
     points",
  "sql": "SELECT ... FROM ... WHERE ...;"
}
```

Listing 4.3: Step 3 prompt — final version

An additional mechanism that contributes to the robustness of the system is the automatic SQL repair functionality implemented in repair_sql(). When the first SQL generation attempt produces a query that fails to execute against the database (due to syntax errors, invalid column references, or incorrect join conditions) the system does not immediately report a failure. Instead, it automatically invokes repair_sql(), passing the original natural language question, the failed SQL query and the database error message to the model. The model receives the same full system prompt used for generation, enriched with the specific error context and is asked to produce a corrected version of the query. The repaired SQL is then validated and executed again. If the repair succeeds, the output file is saved with the repaired flag set to true and the original failed SQL stored for reference. If the repair also fails, the error is propagated to test_question.py and the question is marked as failed for that cycle.

This two-attempt strategy (generate, then repair on failure) reduces the number of execution errors that would otherwise count as mismatches simply because of recoverable syntactic issues.

These fixes produced the best result observed across all prompt versions. Crucially, the final prompt was then applied without any modification to GPT-5 and Gemini 2.5 Pro, two architectures fundamentally different from Claude Opus 4, on which the prompt was developed. All three models converged to approximately 86% execution accuracy, demonstrating that the prompt design is robust across different model architectures and that the domain-specific rules encoded in the prompt are sufficiently general to guide any state-of-the-art Large Language Model toward correct SQL generation on this database.

Step	Main additions	Matches	Accuracy	Std Dev
Step 1	Dynamic schema, minimal rules	26/44	59.09%	–
Step 2	Aggregation, date, structure rules	31.40/44	71.36%	±2.53%
Step 3	GT-verified fixes	38.10/44	86.59%	±0.68%

Table 4.2: Claude Opus 4 prompt engineering progression — mean execution accuracy over 10 cycles.

The progression from 59% to a mean of 86.59% ± 0.68% represents an improvement of more than 27 percentage points, achieved exclusively through prompt engineering without any fine-tuning or modification of the underlying model weights. The standard deviation decreases significantly between Step 2 and Step 3, from ±2.53% to ±0.68%, indicating that the final prompt not only achieves higher accuracy but also produces more consistent results across independent evaluation cycles, with less variability due to the non-deterministic nature of LLM generation.

4.4 Experimental Design

The benchmark consists of 55 natural language questions manually designed to cover the main query types that a production manager or shop-floor operator would typically make to a MES system. The questions include simple lookups on a single table, complex multi-table joins, aggregations, date range filters and domain-specific queries involving status values, working group assignments and material allocations. Of the 55 questions, 11 correspond to write operations (INSERT, UPDATE, or DELETE) which cannot be evaluated through result set comparison and are therefore excluded. The effective benchmark consists of 44 comparable questions per cycle.

For each model, the evaluation was run for $N = 10$ independent cycles. This choice is motivated by the non-deterministic nature of LLM generation: the same prompt may produce slightly different SQL queries across different API calls, so a single evaluation cycle is not sufficient to distinguish a genuine prompt deficiency from a random generation error. Running 10 cycles and computing the mean and standard deviation of accuracy provides a more reliable estimate of the true performance of the system. The total number of queries submitted to each model is therefore:

$$N \times Q = 10 \times 55 = 550 \text{ queries} \tag{4.4}$$

for a grand total of 1,650 evaluations across the three models.

The prompt engineering process was carried out iteratively using Claude Opus 4 as the reference model, progressively refining the prompt through the three steps described in Section 4.3. Once the prompt reached approximately 86% execution accuracy on Claude Opus 4, it was applied without any modification to GPT-5 and Gemini 2.5 Pro, running the same 10-cycle evaluation on each model. This approach allows measuring how robust the prompt design is across fundamentally different model architectures. The implementation details are described in Chapter 5 and the results are presented in Chapter 6.

Implementation

This chapter describes in detail the implementation of the Text-to-SQL pipeline developed in this work. Each of the five Python files that compose the system is analyzed individually, explaining the role of every function, the data it receives as input, the operations it performs and the output it produces.

5.1 run_evaluation_cycles.py

This file serves as the orchestration layer of the benchmark system. It doesn't perform any LLM interaction or database query directly, its only responsibility is to coordinate the repeated execution of the pipeline, collect the results of each cycle and produce a statistical report. The entry point is the main() function, which controls the entire execution flow.

The `main()` function accepts three command-line arguments:

- - - cycles: sets the number of evaluation cycles to run (default 10);
- - - model: specifies the LLM to use, identified by its API model string (default claude-opus-4-7);
- - - eval-only: a boolean flag that controls whether the output files are regenerated at each cycle. When set to False (default), the pipeline invokes test_question.py at every cycle, regenerating all 55 output files before running the evaluation. When set to True, the generation step is skipped and Evaluation.py is run directly on the existing output files, which is practically useful when the outputs have already been generated and only the accuracy needs to be recomputed.

Before entering the main loop, the function initializes three data structures:

- all_metrics is a list that will accumulate one dictionary of numeric metrics per cycle.
- cycle_reports is a list of per-cycle records, each containing the cycle number, a timestamp, the metrics dictionary and the list of mismatched question IDs for that cycle.
- mismatch_freq is a Counter object that maps each question ID to the number of cycles in which it produced an incorrect result.

For each cycle, the function performs two sequential steps. In the first step, unless –eval-only is active, it calls run_script() to invoke test_question.py as a subprocess, passing the model name as an argument. This triggers SQL generation and response writing for all 55 questions. A two-second pause is inserted after this step to avoid hitting API rate limits. In the second step, it calls run_script() again to invoke Evaluation.py, which compares the generated SQL results against the ground truth and writes the evaluation report. The standard output of the evaluation subprocess is then passed to parse_mismatch_ids() to extract the IDs of the questions that failed in that cycle and the mismatch_freq counter is updated.
After the evaluation step, the function calls load_report() to read the JSON report written by Evaluation.py and extract_metrics() to flatten it into a dictionary of numeric values. This dictionary is appended to all_metrics together with a timestamped cycle record in cycle_reports.
After all cycles have completed, the function calls compute_stats() to compute the aggregate statistics and print_summary() to print the final report to the console. The complete results are then serialized to cycles_report.json, which includes the timestamp, the model name, the number of cycles run, the per-cycle detail records, the aggregate statistics, the raw accuracy values per cycle and the mismatch frequency dictionary sorted by descending failure count.

run_script(script, extra_args) is a utility function that executes a given Python script as a subprocess. It builds the command to run by combining the path of the virtual environment Python interpreter, the path of the script to execute and any additional arguments. The command is then launched using subprocess.run(), which captures both the standard output and the standard error so that they can be printed to the console and used by the caller. The function returns a pair of values: a boolean that indicates whether the script completed successfully (return code 0) and a string containing the full output produced by the script. This output is used by main() to extract the list of mismatched question IDs after each evaluation step.

parse_mismatch_ids(eval_stdout) function parses the standard output of Evaluation.py to extract the identifiers of questions that produced a mismatch in a given cycle. It iterates line by line over the output string and selects only those lines that start with the tag [MISMATCH]. For each matching line, it splits the line on whitespace and extracts the second token, which contains the filename of the failing question (e.g. `2.json`), then strips the .json extension to obtain the numeric question ID. The result is a list of string IDs corresponding to all questions that failed in that cycle, which is used by main() to update the mismatch_freq counter.
load_report() reads and deserializes the evaluation_results.json file written by Evaluation.py after each evaluation cycle, returning its content as a Python dictionary.

extract_metrics(report) takes this dictionary and extracts a fixed set of numeric fields using dict.get() with a default value of 0, so that missing fields do not raise exceptions. The extracted fields include total_files, comparable_queries, matches, mismatches, execution_accuracy and the various skip and error counters. The result is a flat dictionary representing the performance of the model on the 55 questions in that specific cycle, which is appended to all_metrics by main().

compute_stats(all_metrics) takes the list of per-cycle metric dictionaries and computes, for each metric, the mean, population standard deviation, minimum and maximum across all cycles. The mean $\bar{A}$ and standard deviation σ are computed directly in pure Python without external statistical libraries:

$$\bar{A} = \frac{1}{N} \sum_{i=1}^{N} A_i \tag{5.1}$$

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(A_i - \bar{A}\right)^2} \tag{5.2}$$

where A_i denotes the execution accuracy measured in cycle i and N is the total number of completed cycles. If only one cycle has been completed, the variance is set to 0.0 to avoid division by zero. The result is a nested dictionary mapping each metric name to a sub-dictionary containing the raw per-cycle values, the mean, the standard deviation, the minimum and the maximum.

print_summary(stats, n_cycles, mismatch_freq) formats and prints the final statistical report to standard output. For each metric it prints the mean

and standard deviation. It then prints the per-cycle accuracy values and the min/max range. The most analytically relevant part of the output is the mismatch frequency analysis: the function classifies each question in mismatch_freq into one of three categories based on its failure rate across cycles. Questions that fail in every cycle are labeled always fail. Those that fail in at least 70% of cycles are labeled often fail. The remaining ones are labeled sometimes fail. This three-tier classification was the primary instrument used during the prompt engineering process to identify which questions required targeted fixes: questions in the always fail category represented structural mismatches between the model's SQL generation behavior and the expected output and were systematically analyzed against the ground truth to identify the root cause of each failure.

Listing 5.1 shows the final aggregate report printed by print_summary() after 10 complete cycles using Claude Opus 4 as the underlying model. The report includes the mean and standard deviation for every metric, the per-cycle accuracy values and the mismatch frequency classification.

```
  Total files           : 55.00  (+-0.00)
  Comparable queries    : 44.00  (+-0.00)
  Matches               : 38.10  (+-0.30)
  Mismatches            : 5.90  (+-0.30)
  Execution accuracy    : 86.59%  (+-0.68%)
  --- Skipped ---
  No GT SQL             : 0.00  (+-0.00)
  GT narrativo          : 0.00  (+-0.00)
  Write ops (escluse)   : 11.00  (+-0.00)
  No generated SQL      : 0.00  (+-0.00)
  No output file        : 0.00  (+-0.00)
  --- Errors ---
  GT exec errors        : 0.00  (+-0.00)
  Gen exec errors       : 0.00  (+-0.00)

  Accuracy per cycle    : [86.36%, 86.36%, 86.36%, 88.64%,
      86.36%,
                           86.36%, 86.36%, 86.36%, 86.36%,
                               86.36%]
  Min accuracy          : 86.36%
  Max accuracy          : 88.64%

  MISMATCH FREQUENCY (su 10 cicli)
  Always failure (10/10)  : Q2, Q10, Q28, Q35, Q54
  often failure (>=70%)   : Q29(9)
```

Listing 5.1: Final aggregate report after 10 evaluation cycles (Claude Opus 4)

These results show that the system achieved a mean execution accuracy of 86.59% ± 0.68% across 10 cycles, with a minimum of 86.36% and a maximum of 88.64%. Five questions — Q2, Q10, Q28, Q35 and Q54 — failed in every single cycle, indicating a structural mismatch between the model's interpretation of these questions and the expected ground truth output rather than a stochastic generation error. Question Q29 failed in 9 out of 10 cycles, suggesting a borderline case where the model occasionally produces the correct result. This per-question failure frequency analysis was the primary diagnostic tool used throughout the iterative prompt engineering process described in Section 4.3, as it distinguishes genuine prompt deficiencies from the natural non-determinism of LLM-generated SQL.

The summary report produced at the end of each run contains several categories of results, each reflecting a different outcome of the evaluation process. Understanding these categories is important to correctly interpret the execution accuracy figure.

Comparable queries (44) represents the number of questions that were actually evaluated, that is, the subset of the 55 questions for which a meaningful comparison between the generated SQL and the ground truth was possible. The remaining 11 questions were excluded for the reasons described below.

Matches (38.10 ± 0.30) is the average number of questions per cycle for which the generated SQL produced a result set that matched the ground truth. The small standard deviation of 0.30 confirms that the system behaves consistently across cycles, with only occasional variation due to the non-deterministic nature of LLM generation.

Mismatches (5.90 ± 0.30) is the average number of questions per cycle for which the generated SQL was executed successfully but produced a result set that did not match the ground truth. These are the cases that directly affect the execution accuracy and that were the focus of the iterative prompt engineering process.

Execution accuracy (86.59% ± 0.68%) is the primary evaluation metric, computed as the ratio of matches to comparable queries:

$$A = \frac{\text{matches}}{\text{matches} + \text{mismatches}} = \frac{38.10}{44} \approx 86.59\% \tag{5.3}$$

The standard deviation of 0.68% confirms the stability of the result across the 10 independent cycles.
No GT SQL (0) counts the questions for which the ground truth field in the input JSON file is empty or missing. These questions cannot be evaluated since there is no reference SQL to compare against. In this benchmark all 55 questions have a ground truth, so this counter is always 0.
GT narrativo (0) counts the questions for which the ground truth field contains a natural language description rather than a valid SQL query. This can occur in datasets where some answers are not expressible as a single SELECT statement and were instead described in plain text. The evaluation module detects this case by checking whether the ground truth string starts with SELECT or WITH and skips the question if it does not. In this benchmark no such cases exist.
Write ops (11) counts the questions that correspond to write operations — INSERT, UPDATE, or DELETE — and are therefore excluded from evaluation. These questions are marked with the field evaluable: false in the input JSON file, since they modify the database state rather than returning a result set, making a value-based comparison against a ground truth impossible. Out of the 55 questions in the benchmark, 11 are write operations and are consistently skipped in every cycle.
No generated SQL (0) counts the questions for which the output file exists but does not contain a generated SQL query, for example because the LLM returned a malformed response that could not be parsed. In this benchmark this counter is always 0, meaning that the JSON extraction mechanism in llm.py successfully recovered a valid SQL from every model response.
No output file (0) counts the questions for which no output file was found in the Output/ directory, meaning that test_question.py failed to process that question entirely. In this benchmark this counter is always 0.
GT exec errors (0) counts the questions for which the ground truth SQL raised an exception when executed against the database. This would indicate an error in the benchmark itself rather than in the system and in this case it never occurs.
Gen exec errors (0) counts the questions for which the generated SQL raised a database exception during evaluation, after the repair mechanism in fashion.py had already been attempted. A non-zero value here would indicate that the repair step failed to produce an executable query. In this benchmark this counter is always 0, confirming that the combination of initial generation and automatic repair consistently produces syntactically valid SQL for every question.

Run_evaluation_cycles.py coordinates the repeated execution of the pipeline

across multiple evaluation cycles by sequentially calling test_question.py and Evaluation.py, collecting metrics and mismatch frequencies for each cycle and producing a statistical report.

In order to obtain a statistically reliable estimate of execution accuracy for each model, the evaluation pipeline was run for $N = 10$ independent cycles (variable N_CYCLES set to 10). Since each cycle processes the full benchmark of $Q = 55$ natural language questions, the total number of queries submitted to the system for each model amounts to 550.

5.2 test_question.py

This file is the core processing script of the pipeline. Its main() function iterates over all 55 input questions one by one, coordinates the SQL generation and execution through fashion.py, generates the natural language response through llm.py and saves the result for each question to a JSON output file.

The script accepts a single command-line argument, –model, which specifies the LLM to use for both SQL generation and response generation. This argument is passed by run_evaluation_cycles.py when it invokes the script as a subprocess, so the model is always set externally and consistently across all 55 questions of a given cycle.

The main() function starts by defining the paths of the input and output directories and creating the output folder if it does not already exist. It then enters a loop over the 55 question files, processing them one at a time.
For each question, the function first reads the input JSON file and extracts the natural language query from the q_en field. It then calls get_data() from fashion.py, passing the query and the model name. This function handles the entire SQL generation, validation and database execution process and returns a JSON string containing the generated SQL, the result rows, the execution time, the reasoning trace produced by the model, the token usage counters and a flag indicating whether the SQL required an automatic repair step.
Once the data is available, the function calls generate_response() from llm.py to translate the SQL result into a natural language answer. The call provides the model with a structured prompt containing the generated SQL query and the result rows, wrapped in Context tags, followed by the original user question. The system prompt instructs the model to answer strictly on the basis of what the SQL returned, without inventing information and to state that no data was found if the result set is empty.

Finally, the function assembles the output dictionary by taking the original input file, removing the ground truth SQL field so that it cannot influence the evaluation and adding the generated SQL, the result rows, the natural language answer, the reasoning trace and the token counters. The assembled dictionary is saved as a JSON file in the Output/ directory, using the same numeric filename as the input.
The entire loop is wrapped in a try/except block that catches missing input files, malformed JSON and any other unexpected error, printing a [FAIL] or [SKIP] message to the console and continuing with the following question without interrupting the cycle.

Each input question is stored as a JSON file containing the numeric identifier, the natural language question in English, the original Chinese formulation, the ground truth SQL query used for evaluation and a steps field inherited from the original Chat with MES framework that describes the multi-step execution plan used by the CWM pipeline. An example is shown in Listing 5.2.

```
{
  "ID": 1,
  "q_en": "What are the orders for PolyU?",
  "gt": "SELECT o.id, o.order_name FROM orders o
         JOIN customers c ON o.user_id = c.id
         WHERE c.customer_name = 'PolyU';",
  "steps": [
    {
      "type": "SQL",
      "sql": "SELECT id FROM customers
              WHERE customer_name = 'PolyU';",
      "description": "Retrieve the customer ID for PolyU
         ",
      "results": [{"id": 1}]
    },
    {
      "type": "SQL",
      "sql": "SELECT id, order_name FROM orders
              WHERE user_id = 1;",
      "description": "Retrieve all orders for PolyU",
      "results": [
        {"id": 1, "order_name": "PolyU_Tshirt"},
        {"id": 9, "order_name": "PolyU_Skirt"}
      ]
```

```
    }
  ]
}
```

Listing 5.2: Example of an input question file

The gt field contains the ground truth SQL query used by Evaluation.py to verify the correctness of the generated query. This field is removed from the output file before saving, so that it remains strictly confined to the input and cannot influence the generation process. The steps field contains the multi-step execution plan produced by the original CWM pipeline and is not used by the system presented in this work, which generates a single SQL query directly from the natural language question. In the present system, the evaluable field was added to mark write operation questions that must be skipped during evaluation.

After processing, the system produces a corresponding output file for each question, shown in Listing 5.3. The file is a JSON dictionary that extends the original input file with the results of the processing pipeline.

```
{
  "ID": 1,
  "q_en": "What are the orders for PolyU?",
  "plan": "1. Which tables are needed? ...",
  "repaired": false,
  "sqls": [
    {
      "command": "SELECT o.id AS order_id, pt.description
         ,
        p.attributes FROM customers c JOIN orders o ON
        c.id = o.user_id ... WHERE c.customer_name = '
           PolyU';",
      "execution time": "0.0013",
      "Result of executing SQL": [
        {"order_id": 1, "description": "T-shirt",
         "attributes": "Red, M size"},
        {"order_id": 9, "description": "Skirt",
         "attributes": "Pink, M size"}
      ]
    }
  ],
  "answer": "The orders for PolyU are:\n- Order ID: 1\n-
     Order ID: 9",
  "tokens": {
    "completion_tokens": 407,
```

```
    "prompt_tokens": 4783,
    "total_tokens": 5517
  }
}
```

Listing 5.3: Example of an output file produced by test_question.py

The output file contains the following fields:
ID and q_en are inherited directly from the input file and identify the question. q_ch contains the original Chinese formulation of the question, as the benchmark was originally designed for a Chinese manufacturing context.

plan contains the step-by-step reasoning trace produced by the model before generating the SQL query. It reflects the six mandatory reasoning steps encoded in the prompt, table selection, join identification, filter definition, column selection, domain terminology check and query type classification and provides full transparency into the model's decision process for each question.

repaired is a boolean flag that indicates whether the first SQL query generated by the model failed to execute and had to be automatically repaired by repair_sql(). When true, two additional fields are present: original_sql, which stores the failed query and repair_error, which stores the database error message that triggered the repair.

sqls is a list containing a single dictionary with three fields: command, which is the final SQL query that was executed against the database; execution time, which is the time in seconds taken by the database to return the result; and Result of executing SQL, which contains the actual rows returned by the database, each represented as a dictionary mapping column names to values.

answer is the natural language response generated by the model based on the SQL result. It is the final output that would be shown to the user in a real deployment of the system.

tokens is a dictionary containing the token usage statistics for the SQL generation step: completion_tokens is the number of tokens in the model response, prompt_tokens is the number of tokens in the input prompt including the full database schema and total_tokens is their sum. These counters are useful for estimating the cost of running the system at scale and for comparing the prompt length across different prompt versions.

Note that the ground truth SQL field (gt), which is present in the input

file, is explicitly removed before saving the output. This ensures that the evaluation module, which loads both the input and the output file, always retrieves the ground truth from the input file and the generated SQL from the output file, keeping the two strictly separate.

5.3 fashion.py

This file sits between test_question.py, which calls it to retrieve data for each question and llm.py, which it calls to generate and repair SQL queries. Its job is to receive a natural language question, ask llm.py to generate the corresponding SQL query, execute it against the MySQL database and return the results. If the first execution attempt fails due to a database error, the file automatically asks llm.py to repair the query and tries again before giving up. The file exposes a single public function, get_data(), supported by two internal helper functions that handle SQL validation and date serialization.

validate_sql(sql) receives a string containing a SQL query and verifies that it is valid to execute. It checks that the string is non-empty and that it starts with either SELECT or WITH, enforcing a strict read-only policy. If either check fails, the function raises an error with a descriptive message. If both checks pass, the query string is returned. This function is called before every execution attempt, both for the initially generated query and for any repaired version, to ensure that the system never executes write operations against the database.

datetime_to_str(o) is used during JSON serialization. MySQL returns date and time columns as Python datetime objects, which cannot be serialized to JSON natively. This function converts any datetime object it receives into a formatted string of the form YYYY-MM-DD HH:MM. It is passed as the default serializer to json.dumps(), so it is called automatically whenever the serializer encounters a type it cannot handle. For any other type it returns nothing, which causes the serializer to raise an error as expected.

get_data(query, model, retry, return_sql) is the main function of the file. It accepts the natural language question as a string, the model name to use for SQL generation, a retry flag that controls whether an automatic repair is attempted on failure (default 1, meaning one repair attempt is allowed) and a return_sql flag that controls the format of the returned data.
The function starts by reading the database connection parameters from the configuration file and then proceeds through the following steps.

First, it calls generate_sql() from llm.py, passing the natural language query and the model name. This function submits the query to the selected LLM and returns a dictionary containing the generated SQL and the reasoning trace produced by the model. The function checks that the output is valid and passes the SQL string to validate_sql() to confirm it is a read-only query.
Second, it opens a connection to the MySQL database using pymysql, configuring the cursor to return each result row as a dictionary mapping column names to values rather than as a plain tuple. This makes the output easier to serialize and interpret.
Third, it executes the validated SQL query and measures the execution time. If the execution succeeds, it assembles a result dictionary containing the SQL query, the result rows, the execution time, the reasoning trace, the token usage counters and a repaired flag set to false.
If the execution raises a database exception and the retry flag is greater than zero, the function attempts an automatic repair. It calls repair_sql() from llm.py, passing the original question, the failed SQL and the error message. The repaired SQL is validated and executed again. If the repair succeeds, the result dictionary is assembled as before but with the repaired flag set to true and the original failed SQL and the error message are stored as additional fields. The token counts are summed across both the initial generation call and the repair call.

Finally, the full result dictionary is serialized to a JSON string using datetime_to_str to handle any datetime values in the result rows and returned together with an empty error string.
The get_data() function is designed to never crash the program. Instead of letting an error propagate upward and interrupt the entire evaluation cycle, it catches any exception internally and handles it. The function always returns two values: the retrieved data and an error string. If everything works properly, it returns the data and an empty error string. If something goes wrong at any point, it returns an empty string in place of the data and the error message as the second value, allowing test_question.py to detect the failure simply by checking whether the error string is empty or not.

The database connection is always closed in a finally block, regardless of whether the execution succeeded or failed, to prevent connection leaks.

5.4 llm.py

This file is the module responsible for all interactions between the system and the Large Language Models. It provides a single, unified interface that works identically regardless of which model is selected, GPT-5, Claude Opus 4, or Gemini 2.5 Pro, hiding the differences between the three APIs behind a common set of functions. It handles provider selection, prompt construction, database schema extraction, JSON parsing and exposes the three public functions that the rest of the system uses to communicate with the models. From the perspective of this thesis, this file represents the true core of the work: it is here that the system prompt given to the LLM is defined and constructed, encoding all the domain knowledge, SQL generation rules and reasoning requirements that were iteratively refined throughout the prompt engineering process described in Section 4.3. The quality and precision of this prompt is the primary factor that determines the execution accuracy of the entire system.

The file defines a class hierarchy to manage the interaction with the three supported LLM providers. A base class, BaseProvider, declares a single abstract method generate(system, user, max_tokens) that all concrete provider classes must implement, ensuring that the rest of the system can call the same method regardless of which model is in use.
AnthropicProvider is initialized with the model name string and instantiates the Anthropic Python client using the API key read from the configuration file. Its generate() method calls client.messages.create() with the model name, the system and user prompts and the token limit, then extracts the response text and the token usage from the response object.
OpenAIProvider is initialized with the model name string and instantiates the OpenAI Python client using the API key and an optional base URL from the configuration file, which allows the use of custom API endpoints compatible with the OpenAI interface. Its generate() method calls client.chat.completions.create(), passing the system prompt as a message with role system and the user prompt as a message with role user, then extracts the response text and token usage from the completion object.
GeminiProvider is initialized with the model name string and instantiates the Google GenAI client using the new google.genai SDK and the API key from the configuration file. Its generate() method calls client.models.generate_content(), passing the model name, the user prompt as contents and a GenerateContentConfig object that carries the system instruction and the maximum output token count. Token usage is extracted from re-

sponse.usage_metadata, with a fallback to 0 for fields that may be None when usage metadata is not available.
All three generate() methods return the same structure: a pair of values where the first is the model response as a plain string and the second is a dictionary containing prompt_tokens, completion_tokens and total_tokens. The selection of the appropriate provider is handled by an internal factory function, _get_provider(model), which takes the model name string and inspects its prefix to decide which provider class to instantiate. Names starting with claude are routed to AnthropicProvider, names starting with gpt, o1, or o3 are routed to OpenAIProvider and names starting with gemini are routed to GeminiProvider. If the prefix does not match any known pattern, the function raises an error with an instructive message suggesting how to extend the routing logic to support additional models.
An internal helper function, _db_conn(), reads the database connection parameters from the configuration file and returns an open connection to the MySQL database, configured to return result rows as dictionaries rather than plain tuples.

build_schema_markdown() connects to the database and queries the MySQL information_schema.COLUMNS to retrieve the structure of all tables, column names, data types, nullability and key constraints, ordered by table name and column position. For each column it builds a descriptive line containing the column name, data type, nullability, key type and any extra attributes such as auto_increment. The columns are grouped by table and the result is formatted as a Markdown string where each table is introduced by a heading followed by one bullet point per column. The database connection is always closed in a finally block. An excerpt of the output produced for the garment MES database is shown in Listing 5.4.

```
## customers
- id int(10) NOT NULL KEY auto_increment
- customer_name varchar(255) NOT NULL

## orders
- id int(10) NOT NULL KEY auto_increment
- user_id int(10) NOT NULL KEY
- order_name varchar(255) NULL
- created_at timestamp NULL

## cutting_tasks
- id int(10) NOT NULL KEY auto_increment
- start_time timestamp NULL
```

```
- end_time timestamp NULL
- working_group_id int(10) NOT NULL KEY
- planned_number int(11) NULL
- completed_number int(11) NULL
- status int(11) NULL
```

Listing 5.4: Excerpt of the schema injected into the prompt by build_schema_markdown()

build_foreign_keys_markdown() queries the MySQL information_schema.KEY_COLUMN_USAGE view to retrieve all foreign key relationships defined in the target database, filtering out primary key constraints where the referenced table is NULL. For each foreign key it builds a line of the form table.column -> referenced_table.referenced_column. The database connection is always closed in a finally block.

An excerpt of the output for the garment MES database is shown in Listing 5.5.

```
- cutting_material_allocation.cutting_task_id ->
   cutting_tasks.id
- cutting_material_allocation.warehouse_material_id ->
   warehouse_material.id
- cutting_tasks.working_group_id -> working_group.id
- order_product.order_id -> orders.id
- order_product.product_id -> products.id
- orders.user_id -> customers.id
- sewing_tasks.order_product_id -> order_product.id
- sewing_tasks.working_group_id -> working_group.id
```

Listing 5.5: Excerpt of the foreign key relationships injected into the prompt

make_generate_sql_prompt() assembles the full system prompt that is sent to the LLM for every SQL generation call. It starts by calling build_schema_markdown() and build_foreign_keys_markdown() to obtain the current database structure, then combines them with a set of domain-specific sections into a single string. The prompt is organized into seven sections. The CONTEXT section identifies the model as a MySQL expert for a clothing production MES and specifies the database name. The DATABASE SCHEMA section contains the full schema produced by build_schema_markdown(). The FOREIGN KEY RELATIONSHIPS section contains the foreign key list produced by build_foreign_keys_markdown(). The DOMAIN GLOSSARY section contains a set of explicit rules covering status value semantics, task table disambiguation, standard column selection

for task listing queries, progress percentage patterns, material query patterns, warehouse name conventions, aggregation null prevention, date range rules and the prohibition of CONCAT and GROUP_CONCAT in inventory queries — each rule specifying the exact SQL pattern to use for a given class of natural language question. The SQL RULES section contains general constraints applicable to all queries, including the prohibition of write operations and the requirement to use only tables and columns present in the schema. The REASONING REQUIREMENT section contains a mandatory chain-of-thought instruction requiring the model to reason through six explicit steps before writing any SQL: identifying the required tables, determining the necessary joins, identifying the applicable filters, selecting the correct output columns, applying domain-specific terminology rules and classifying the query type. Finally, the RESPONSE FORMAT section instructs the model to return a valid JSON object with exactly two keys: thought for the reasoning trace and sql for the generated query, with no markdown fences or additional text. The complete prompt used in the final version of the system is reported in Section 4.3.

_extract_json(text) is an internal helper that takes the raw text output produced by the LLM and attempts to extract a valid JSON object from it. It tries three successive strategies: a direct json.loads() call, a second attempt after stripping any markdown code fences using a regular expression and a final attempt using re.search() to locate the first JSON object in the text. If all three attempts fail, it raises an error with the first 500 characters of the response for debugging purposes. This three-step approach is necessary because different models occasionally wrap their JSON output in markdown code blocks or add explanatory text around it, as shown in Listing 5.6.

````
Here is the SQL query for your request:

```json
{
  "thought": "1. Tables needed: orders, customers...",
  "sql": "SELECT o.* FROM orders o JOIN customers c
          ON o.user_id = c.id
          WHERE c.customer_name = 'PolyU';"
}
```
````

Listing 5.6: Example of a raw model response wrapped in markdown fences

In this case the first json.loads() attempt fails because of the surrounding

text and the markdown fences. The second attempt succeeds after the fences are stripped by the regular expression. The third attempt using re.search() would only be needed if the model added additional text after the closing brace of the JSON object.
generate_sql(query, model) is the first of the three public functions of the file. It takes the natural language query and the model name, selects the appropriate provider via _get_provider() and calls provider.generate() with the system prompt produced by make_generate_sql_prompt() and the natural language query as the user message. The raw text response is passed to _extract_json() to obtain the structured dictionary and the token usage is merged into the result. Any exception is caught and re-raised with a descriptive message including the model name. The function returns a dictionary containing the thought field with the reasoning trace, the sql field with the generated query and the three token counters.
repair_sql(query, failed_sql, error_message, model) is called automatically by fashion.py when the first SQL execution attempt fails. It takes the original natural language question, the failed SQL string, the database error message and the model name. It formats an internal repair prompt template by substituting these three values, then calls provider.generate() with the repair prompt as the user message and the full system prompt as the system message, so that the model has access to all domain rules during the repair attempt. The response is parsed with _extract_json() and validated to ensure the sql key is present. The function returns a dictionary containing the thought field, the repaired sql and the token counters.
generate_response(user_query, system_query, model) generates the final natural language answer from the SQL result. It takes a structured user prompt containing the generated SQL and the result rows wrapped in Context tags, together with a system prompt that instructs the model to answer strictly from the SQL result without inventing information. Unlike generate_sql() and repair_sql(), this function does not attempt to parse the response as JSON, it treats the model output as plain natural language text and returns it directly. Token usage is not tracked for this call in the current implementation.

5.5 Evaluation.py

This file implements the evaluation stage of the pipeline. It does not interact with any LLM, its only job is to load the output files produced by test_question.py, execute both the generated SQL and the ground truth SQL against the database, compare the results and produce a structured

evaluation report. The file is organized around a main() entry point supported by a set of utility functions described below.

load_json(path) opens a JSON file from the given path and returns its content as a Python dictionary.

get_db_connection() reads the database connection parameters from the configuration file and returns an open connection to the MySQL database, configured to return result rows as dictionaries.

execute_sql(sql) takes a SQL query string, opens a database connection, executes the query and returns all result rows together with an error string. If the execution raises an exception, the function catches it and returns an empty list of rows paired with the error message, following the same pattern used in fashion.py. The database connection is always closed in a finally block.

normalize_value(value) converts a single value returned by the MySQL cursor into a normalized string representation suitable for comparison. Datetime objects are formatted as strings in the format YYYY-MM-DD HH:MM. Decimal objects — which MySQL returns for aggregated expressions such as SUM, AVG and computed percentages — are converted to float, rounded to 4 decimal places and returned as strings. None values are kept as None. All other values are converted to string using str(). This normalization is critical to avoid false mismatches caused by type differences between the ground truth and the generated result sets — for example when one query returns a Decimal and the other returns an integer representing the same numeric value.

normalize_rows(rows) applies normalize_value() to every value in every row of a result set. Each row is also sorted by column name so that differences in column ordering between the ground truth and the generated query do not affect the comparison. The resulting list is then sorted lexicographically by the JSON representation of each row, so that differences in row ordering do not affect the comparison either.

row_to_valueset(row) converts a single result row into a frozenset of normalized values, discarding the column names entirely. This is the key step that makes the comparison value-based rather than column-name-based.

rows_to_multiset(rows) applies row_to_valueset() to each row in a result

set and returns the resulting frozenset objects as a sorted list, one per row. This sorted list is what compare_results() uses to perform the final comparison.

compare_results(gt_rows, pred_rows) is the core comparison function. It first checks that the two result sets have the same number of rows, if they differ, it immediately returns False. It then converts both result sets to multisets of value sets using rows_to_multiset() and iterates over the paired rows. For each pair, it checks whether the ground truth value set is a subset of the predicted value set. This subset check rather than strict equality is a deliberate design choice: it allows the generated SQL to return additional columns beyond those required by the ground truth, while still correctly detecting mismatches when a required value is missing or wrong. If any pair fails the subset check, the function returns False.

get_gt_sql(input_data) extracts and validates the ground truth SQL from an input question file. It first checks whether the evaluable field is set to False, which marks write operation questions that have no ground truth SQL and must be skipped. It then checks that the gt field is a non-empty string starting with SELECT or WITH — if it contains narrative text instead of a SQL query, the question is also skipped. The function returns a pair of values: the ground truth SQL string and a skip reason string that is empty on success and contains a descriptive code when the question must be skipped.

get_generated_sql(output_data) extracts the generated SQL from an output file. It retrieves the command field from the first entry of the sqls list and applies a regular expression to remove any Markdown hyperlinks of the form [alias](url) that may have been introduced by older versions of the system, replacing them with just the alias text. The cleaned SQL string is returned after stripping whitespace.

evaluate_pair(input_file, output_file) orchestrates the evaluation of a single question. It loads both the input and the output file, extracts the ground truth SQL via get_gt_sql() and the generated SQL via get_generated_sql() and returns the appropriate skip reason immediately if either is missing or invalid. If both queries are available, it executes them both against the database via execute_sql() and checks for execution errors. If both execute successfully, it calls compare_results() to determine whether the result sets match. The function returns a pair of values: a status string that is one of match, mismatch, missing_gt, gt_not_sql, not_evaluable_write_op, missing_generated_sql, gt_execution_error, or generated_execution_error; and

an optional dictionary containing diagnostic information such as the row counts and a sample of the first two rows of each result set, used for debugging mismatches.

main() is the entry point of the file. It initializes a set of counters for each possible outcome category and iterates over all 55 input JSON files sorted by numeric filename. For each question, it first checks whether the corresponding output file exists — if not, the question is counted as missing_output. Otherwise, it calls evaluate_pair() and increments the appropriate counter based on the returned status. For each question, a status label is printed to the console — [MATCH], [MISMATCH], [SKIP-WRITE] and so on — followed by the row counts when the status is mismatch. This console output is what run_evaluation_cycles.py captures and parses to extract the IDs of the failing questions after each cycle.

After all questions have been evaluated, main() computes the execution accuracy as:

$$A = \frac{\text{matches}}{\text{matches} + \text{mismatches}} \tag{5.4}$$

where the denominator excludes all skipped and errored questions, so that the accuracy is computed only over the 44 questions that were actually evaluated. The result is rounded to 4 decimal places. The full evaluation report — containing all counters, the execution accuracy and the per-question detail list — is serialized to evaluation_results.json and subsequently read by run_evaluation_cycles.py to update the cycle statistics.

Conclusions

6.1 Summary of the Work

This thesis presented the design, implementation and evaluation of a Text-to-SQL system that enables natural language interaction with a Manufacturing Execution System database. The work was motivated by the growing adoption of Industry 4.0 and Industry 5.0 principles in manufacturing, which push companies to make production data accessible to everyone in the organization, from shop-floor operators to production managers, without requiring technical knowledge of SQL or database structures. This trend is further accelerated by the increasing interest of the global economy in Large Language Models. In line with the human-centric pillar of Industry 5.0 introduced in Chapter 1, the system was designed so that the problem of adaptation falls on the machine rather than on the worker: instead of training operators to navigate predefined forms and fixed dashboards, the system interprets their natural language questions directly, letting the system adapt to the worker rather than the opposite.

The system was built on top of the garment MES database and benchmark proposed by Yuan et al. in the Chat with MES framework [1], which provided 55 natural language questions paired with ground truth SQL queries covering the full range of query types typical of a production management environment. Starting from the original static prompt of the Chat with MES framework, which embedded the database schema as a fixed string hard coded directly into the prompt, the first contribution of this work was the introduction of a dynamic schema injection mechanism, which replaced the static description with a query to the MySQL information_schema. This alone brought the system to approximately 60% execution accuracy while making it automatically adaptable to any future change in the database structure.

The system was then progressively improved by analyzing which questions failed, to produce a correct result and adding targeted rules to the prompt based on the type of error observed. The process consisted of three steps, each introducing a targeted set of domain-specific rules and brought the execution accuracy from approximately 60% to a mean of 86.59% measured across 10 independent evaluation cycles.

The final prompt was then evaluated across 10 independent cycles on three state of the art Large Language Models, GPT-5 (OpenAI), Claude Opus 4 (Anthropic) and Gemini 2.5 Pro (Google), for a total of 1,650 query evaluations. The results confirmed the robustness of the prompt design across all three models: GPT-5 achieved a mean of 88.41% ± 0.68%, Claude Opus 4 achieved a mean of 86.59% ± 0.68% and Gemini 2.5 Pro achieved a mean of 86.36% with zero variance across all 10 cycles (±0.00%), producing an identical result on every run. Table 6.1 summarizes these results. The same 55-question, unmodified prompt allowed all three models to reach very similar accuracy levels, all comfortably above the 80% accuracy reported by the original Chat with MES authors.

Model	**Cycles**	**Mean accuracy**	**Std. dev.**	**Min**	**Max**
GPT-5	10	88.41%	±0.68%	86.36%	88.64%
Claude Opus 4	10	86.59%	±0.68%	86.36%	88.64%
Gemini 2.5 Pro	10	86.36%	±0.00%	86.36%	86.36%

Table 6.1: Execution accuracy across the three evaluated models, aggregated over 10 independent evaluation cycles on the 44 comparable (read-only) benchmark questions.

6.2 Key Aspects

The results of this work lead to four main findings:

- Prompt engineering alone is sufficient to achieve competitive performance. The system surpasses the full Chat with MES pipeline accuracy of 80% reported by the original authors, which relied on additional architectural components including query rewriting and multi-step operation planning. This result demonstrates that a single, well engineered prompt, applied directly to the database without any additional pipeline components, can achieve comparable or superior performance, while being significantly simpler to deploy and maintain.

- The prompt is the bottleneck, not the model. Applying the exact same prompt, without any model-specific tuning, to three fundamentally different model architectures, GPT-5, Claude Opus 4 and Gemini 2.5 Pro, produced execution accuracies of 88.41%, 86.59% and 86.36% respectively. This is the central finding of this work: once the prompt encodes sufficient domain knowledge, the choice of model becomes secondary to prompt quality. GPT-5 and Claude Opus 4 show sub-1% cycle-to-cycle variance ($\pm$0.68%), while Gemini 2.5 Pro produced the exact same result on all 10 cycles. A core set of five questions (Q2, Q10, Q28, Q35 and Q54) fails across all three models in every one of their 10 cycles, pointing to a structural limitation of the prompt design rather than a model-specific weakness. This is likely due to the high complexity of these questions, which require nesting multiple queries to obtain the correct result. Gemini 2.5 Pro also fails consistently (10/10 cycles) on one additional question (Q5), which fails only occasionally for GPT-5 (1/10 cycles) and never for Claude Opus 4.

- Dynamic schema injection makes the system self-updating. By querying the MySQL information_schema at runtime rather than embedding a static schema description in the prompt, the system automatically reflects any structural change to the database, new tables, renamed columns, new foreign key constraints, without requiring any manual update. This property is particularly valuable in real manufacturing environments, where MES databases evolve continuously as new production processes, product categories, or working groups are introduced.

- The deterministic evaluation metric is essential for reproducibility. The use of result set comparison rather than LLM-as-judge evaluation ensures that the accuracy figures reported in this work are fully reproducible: the same generated SQL always produces the same evaluation outcome, regardless of how many times the evaluation is run. This property enabled the statistical analysis across 10 independent cycles, which would not have been meaningful with a non-deterministic evaluation metric. This aspect is also important beyond evaluation itself: querying the MES is typically the first step that precedes further data analysis, so having a standard, consistent output format means that downstream analysis tools can also be automated, making the entire process simpler and faster.

This work makes the following original contributions with respect to the existing literature:

- A fully automated evaluation pipeline that supports repeated, multi-cycle benchmarking of Text-to-SQL systems across multiple LLM providers through a unified, model-agnostic interface. The pipeline supports Claude Opus 4, GPT-5 and Gemini 2.5 Pro through a single codebase without any model-specific modifications.

- A domain-specific prompt for the garment MES database, developed through a systematic, ground-truth-driven engineering process, that encodes explicit SQL generation rules covering status semantics, task table disambiguation, aggregation patterns, material query patterns and response format constraints.

- A dynamic schema injection mechanism that generates the database schema representation at runtime by querying the MySQL information_schema, making the system portable across different databases and automatically consistent with schema changes.

- An empirical demonstration that prompt engineering alone, without fine-tuning, query rewriting, or multi-step operation planning, is sufficient to achieve 86% execution accuracy on a domain-specific MES benchmark, surpassing the 80% reported by the full Chat with MES pipeline.

- An empirical demonstration of model prompt robustness through different models: a prompt developed on Claude Opus 4 retains its effectiveness when applied without modification to GPT-5 (88.41% execution accuracy) and Gemini 2.5 Pro (86.36%), against 86.59% for the reference model, over 10 full evaluation cycles for all three models, providing strong evidence that well-engineered domain-specific prompts generalize across different model architectures.

6.3 Possible Future Work

Several directions for future research emerge naturally from the output of this work.

- The original Chat with MES framework addresses the problem of ambiguous entity references, such as "Hong Kong Polytechnic University" vs "PolyU", through a dynamic vector database that maps natural language entity mentions to their canonical database representations. The current system handles this case through a fixed rule in the prompt, which works for known entities but cannot generalize to new customers,

products, or working groups not anticipated during prompt design. Integrating a lightweight entity disambiguation module, based on embedding similarity or a small lookup table, would make the system more robust to natural language variation without requiring manual prompt updates.

- Multi-step planning for complex queries. The five questions that consistently fail across all three models, Q2, Q10, Q28, Q35 and Q54, share a common characteristic: they require reasoning over multiple intermediate results that cannot be easily expressed as a single SQL query. A multi-step planning component, could decompose these complex questions into sequences of simpler SQL operations, executing them one at a time and passing intermediate results as context to subsequent steps. This approach would potentially close most of the remaining accuracy gap across all three models without requiring further prompt engineering.

- Extension to write operations. The current system is restricted to read-only SELECT queries. Extending it to handle write operations as INSERT, UPDATE and DELETE, would make it a fully functional MES interaction interface rather than a read-only reporting tool. This extension would require a different validation and evaluation strategy, as executing write operations against a live database carries the risk of data corruption. One approach would be to execute write operations against a test database and verify their effects by querying the resulting state.

- Evaluation on other MES databases. The current evaluation is limited to a single garment MES database. Evaluating the system on MES databases from other manufacturing domains, electronics, automotive, pharmaceuticals, would provide a more comprehensive assessment of the generalizability of the prompt engineering approach and the portability of the dynamic schema injection mechanism.

- Fine-tuning comparison. This work demonstrated that prompt engineering alone can achieve competitive performance on a domain-specific Text-to-SQL task. A natural follow-up would be to compare this approach with fine-tuning on the same benchmark, either fine-tuning a smaller open-source model on the 55 question-SQL pairs or using the prompt-generated outputs as synthetic training data for a distillation approach. Such a comparison would quantify the trade-off between the

deployment simplicity of prompt engineering and the potential accuracy gains of model adaptation.

- Real-world deployment. The ultimate validation of this system would be a real-world deployment in an actual manufacturing company, where the system would be exposed to the full diversity of natural language queries from real users with different roles, technical backgrounds and information needs. A user study in this context would provide insights into the usability of the system, the types of queries that users naturally formulate and the gap between the controlled benchmark evaluation and real-world performance.

6.4 Final Considerations

The central message of this work is that the quality of the prompt is the primary determinant of Text-to-SQL performance in domain-specific applications, more so than the choice of the underlying model. A prompt that encodes sufficient domain knowledge as table disambiguation rules, aggregation patterns, column selection constraints and terminology mappings, can guide any state-of-the-art Large Language Model to achieve high execution accuracy on a specialized industrial database, without fine-tuning, without additional pipeline components and without model-specific customization.
This finding has a practical implication for manufacturing companies considering the adoption of natural language interfaces for their MES platforms: the investment should be directed primarily toward prompt design and domain knowledge encoding, rather than toward model selection or infrastructure complexity. A well-engineered prompt is likely to outperform a poorly engineered prompt deployed on the most powerful available model. As Large Language Models continue to improve, the benefits of a well-engineered prompt will also increase.

In the end, this result is also a step toward the human-centric goal of this work: a shop-floor operator or a production manager can access MES data in the same natural, conversational way, no matter which LLM is running behind the system. It is the system, not the worker, that absorbs the complexity of adapting to new technologies. Moreover, both data access and data analysis become much faster: instead of navigating fixed forms or waiting for a technical colleague to write a query, users get an answer directly, in natural language. This is especially valuable in a work environment that increasingly requires fast, data-driven decisions: the faster people can access

and understand data, the faster they can decide.

Bibliography

[1] Zhaolin Yuan et al. "Chat with MES: LLM-driven user interface for manipulating garment manufacturing system through natural language". In: *Journal of Manufacturing Systems* 80 (2025), pp. 1093–1107. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2025.02.008. URL: https://www.sciencedirect.com/science/article/pii/S027861252500038X.

[2] Sabine Pfeiffer. "The Vision of Industrie 4.0 in the Making—a Case of Future Told, Tamed, and Traded". In: *NanoEthics* 11.1 (2017). Epub 2017 Jan 25, pp. 107–121. DOI: 10.1007/s11569-016-0280-3.

[3] Klaus Schwab et al. "The fourth industrial revolution". In: *World Economic Forum*. Vol. 12. 2016.

[4] Malte Brettel et al. "How Virtualization, Decentralization and Network Building Change the Manufacturing Landscape: An Industry 4.0 Perspective". In: *International Journal of Mechanical, Industrial and Aerospace Sciences* 7.0.1 (Nov. 2014). DOI: 10.5281/zenodo.1336426. URL: https://doi.org/10.5281/zenodo.1336426.

[5] Michael Rüßmann et al. *Industry 4.0: The Future of Productivity and Growth in Manufacturing Industries*. Tech. rep. Boston Consulting Group (BCG), Apr. 2015. URL: https://www.bcg.com/publications/2015/engineered_products_project_business_industry_4_future_productivity_growth_manufacturing_industries.

[6] Teresa Dieguez and José Machado. "Reviewing Manufacturing Execution System in Industry 4.0: A Global Approach". In: *EAI Endorsed Transactions on Digital Transformation of Industrial Processes* 1.3 (2025). DOI: 10.4108/dtip.9768. URL: https://publications.eai.eu/index.php/dtip/article/view/9768.

[7] Manon Breque, Lars De Nul, and Athanasios Petridis. *Industry 5.0: Towards a Sustainable, Human-Centric and Resilient European Industry*. Tech. rep. European Commission, Directorate-General for Research and Innovation, 2021. DOI: 10.2777/308407. URL: https://research-and-innovation.ec.europa.eu/knowledge-publications-tools-and-data/publications/all-publications/industry-50-towards-sustainable-human-centric-and-resilient-european-industry_en.

[8] Xun Xu et al. "Industry 4.0 and Industry 5.0—Inception, conception and perception". In: *Journal of Manufacturing Systems* 61 (2021), pp. 530–535. ISSN: 0278-6125. DOI: https://doi.org/10.1016/j.jmsy.2021.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0278612521002119.

[9] Dmitry Ivanov. "The Industry 5.0 framework: viability-based integration of the resilience, sustainability, and human-centricity perspectives". In: *International Journal of Production Research* 61 (Sept. 2022), pp. 1–13. DOI: 10.1080/00207543.2022.2118892.

[10] Andreas Holzinger et al. "Personas for Artificial Intelligence (AI): An Open Source Toolbox". In: *IEEE Access* 10 (2022), pp. 23732–23747. DOI: 10.1109/ACCESS.2022.3154776. URL: https://doi.org/10.1109/ACCESS.2022.3154776.

[11] Wikipedia. *mes*. 2020. URL: https://it.wikipedia.org/wiki/Manufacturing_Execution_System.

[12] sap. *What is a manufacturing execution system (MES)?* URL: https://www.sap.com/uk/products/scm/digital-manufacturing/what-is-mes.html.

[13] Zhou Li, Guwei Li, and Zhengyuan Li. "Application Research of MES in Intelligent Manufacturing Training Factory". In: *Artificial Intelligence and Security*. Ed. by Xingming Sun et al. Cham: Springer International Publishing, 2022, pp. 118–127. ISBN: 978-3-031-06794-5.

[14] Uday Kamath et al. *Large Language Models: A Deep Dive. Bridging Theory and Practice*. 1st ed. Cham: Springer, 2024, pp. XXXIV, 472. ISBN: 978-3-031-65647-7. DOI: 10.1007/978-3-031-65647-7.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[16] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Draft of 3rd edition. 2023. Chap. 8.

[17] Ali Mohammadjafari, Anthony S. Maida, and Raju Gottumukkala. *From Natural Language to SQL: Review of LLM-based Text-to-SQL Systems*. 2025. arXiv: 2410.01066 [cs.CL]. URL: https://arxiv.org/abs/2410.01066.

[18] Zijin Hong et al. *Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL*. 2025. arXiv: 2406.08426 [cs.CL]. URL: https://arxiv.org/abs/2406.08426.

[19] Jinyang Li et al. *Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs*. 2023. arXiv: 2305.03111 [cs.CL]. URL: https://arxiv.org/abs/2305.03111.

[20] Jinqing Lian et al. *ChatBI: Towards Natural Language to Complex Business Intelligence SQL*. 2024. arXiv: 2405.00527 [cs.DB]. URL: https://arxiv.org/abs/2405.00527.

[21] Chenxu Hu et al. *ChatDB: Augmenting LLMs with Databases as Their Symbolic Memory*. 2023. arXiv: 2306.03901 [cs.AI]. URL: https://arxiv.org/abs/2306.03901.

[22] Mohammadreza Pourreza and Davood Rafiei. *DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction*. 2023. arXiv: 2304.11015 [cs.CL]. URL: https://arxiv.org/abs/2304.11015.

[23] Liang Shi et al. "A Survey on Employing Large Language Models for Text-to-SQL Tasks". In: *ACM Comput. Surv.* 58.2 (Sept. 2025). ISSN: 0360-0300. DOI: 10.1145/3737873. URL: https://doi.org/10.1145/3737873.