
Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

Academic Year 2025/2026

Graduation Session March 2026

# Supervised and Contextual Fine-Tuning for Text-to-SQL on Enterprise Databases: An Orchestrated LLM Pipeline

Supervisors:
Prof. Paolo Garza

Candidate:
Marco Pontrandolfo

**Abstract**

Large Language Models (LLMs) have recently demonstrated strong capabilities in natural language understanding and generation, enabling new forms of interaction between users and data systems. Among these applications, text-to-SQL generation represents a particularly relevant use case in enterprise environments, where relational databases remain the backbone of data storage and business analytics. However, directly applying general-purpose LLMs to complex business databases poses significant challenges, including schema complexity, SQL dialect constraints, domain-specific business logic, and cost-efficiency considerations. This thesis investigates the effectiveness of different fine-tuning strategies for adapting LLMs to text-to-SQL tasks in enterprise-like settings. In particular, Supervised Fine-Tuning (SFT) and Contextual Fine-Tuning (Context FT) are systematically compared across multiple configurations. Experiments are conducted using a realistic relational database derived from the Northwind schema, populated with synthetic data to simulate business-scale scenarios. Models are evaluated on a curated set of complex natural language queries, using strict correctness criteria that account for syntactic validity, execution correctness, and adherence to business logic. The study explores incremental fine-tuning strategies, hybrid approaches combining SFT with structured prompting and few-shot learning, and analyzes the trade-offs between accuracy, training cost, and inference cost. Results show that naive supervised fine-tuning with limited data is insufficient for complex SQL generation. However, combining schema-aware fine-tuning with targeted prompting and few-shot demonstrations leads to performance comparable to contextual fine-tuning, while significantly reducing inference token costs. To reflect realistic enterprise deployment scenarios, the fine-tuned model is integrated into a modular orchestrated pipeline based on an orchestrator-worker architecture implemented with LangGraph. In this design, a reasoning model decomposes complex analytical questions into structured sub-queries, delegating SQL generation to the fine-tuned component and synthesizing the results into coherent business reports. This architecture enables scalable interaction with large databases while controlling context window limitations and computational costs. Overall, the findings demonstrate that carefully designed fine-tuning strategies, combined with modular system orchestration, provide a practical and cost-efficient solution for deploying LLM-based text-to-SQL systems in enterprise environments.

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Motivation and Context

### 1.1.1 Description of the Business and Technological Context

Large Language Models (LLMs) have emerged as powerful tools for a wide range of natural language processing tasks, from text generation to question answering and semantic analysis. Their potential extends beyond traditional applications, providing novel opportunities to improve the way businesses interact with data. In particular, the ability to generate Structured Query Language (SQL) queries directly from natural language has significant implications for businesses that rely on relational databases for decision-making and data analytics.

The application of LLMs to generate SQL from natural language queries, known as text-to-SQL, offers a promising solution to the challenges that businesses face in accessing and interacting with their data. Text-to-SQL allows non-technical users to interact with complex databases through simple, intuitive queries, eliminating the need for deep knowledge of SQL syntax or relational database structures. This capability is particularly valuable in business contexts where rapid decision-making is essential, and data-driven insights need to be extracted quickly and efficiently from large datasets.

By enabling natural language queries, LLMs help democratize access to data, empowering a broader range of employees to interact with databases without relying on specialized technical teams. This can lead to improved productivity, better data-driven decision-making, and greater overall business efficiency. Additionally, automating the generation of SQL queries can streamline workflows, reduce the risk of errors in manual query writing, and increase the speed at which data is analyzed.

### 1.1.2 Challenges in the Business Context

Despite the potential of LLMs to revolutionize data access and analysis, their application in business environments presents a number of challenges. One of the main issues is the **complexity of enterprise databases**. Business databases typically contain vast amounts of data organized into multiple interconnected tables, each with its own schema and set of relationships. The structure and complexity of these databases can make it difficult for general-purpose language models to effectively generate accurate SQL queries, especially as the complexity of the queries and the database schema increases.

In addition to schema complexity, there are several other challenges that arise when applying LLMs in a business context. One major concern is data privacy. Businesses often work with sensitive data, and it is essential to ensure that the generated SQL queries do not inadvertently expose confidential information. This requires careful handling of data access permissions and the ability to control what data can be queried by non-technical users.

Another challenge is the lack of domain-specific knowledge in general-purpose LLMs. While these models are highly capable of understanding and generating text in a wide range of contexts, they may lack the domain-specific knowledge necessary to interpret business requirements or understand the specific logic embedded in a company's data. For example, the meaning of terms like "revenue" or "customer" may differ across industries, and this ambiguity can lead to incorrect or incomplete SQL queries. Similarly, business-specific rules—such as how to compute discounts, handle inventory, or aggregate sales data—may not be easily captured by a model that has been trained on general text data.

Finally, there is the challenge of model generalization. While pre-trained LLMs are capable of performing well on a wide range of tasks, they may struggle when faced with the specific nuances of business logic and SQL generation, especially in complex scenarios. A model that has been trained on generic text data may not perform well in generating SQL queries that adhere to the syntax and structure of MySQL or other SQL dialects, or that account for complex relationships between tables in a business database.

### 1.1.3 The Need for a Specialized Approach

Given the challenges outlined above, it is clear that the direct application of general-purpose LLMs to enterprise SQL generation may not be sufficient. A more specialized approach is needed to adapt these models to the specific requirements of business databases and the generation of accurate, contextually appropriate SQL queries.

One solution to this problem is the use of fine-tuning techniques, which allow pre-trained LLMs to be adapted to specific tasks and domains. Fine-tuning involves

training a model on a smaller, task-specific dataset, which helps the model learn the nuances and domain-specific knowledge required for generating SQL queries in a particular context. By fine-tuning a general-purpose model like GPT-4.1 Mini, it is possible to adapt it to understand the structure of a business's database schema, as well as the specific business logic that must be reflected in SQL queries.

Furthermore, fine-tuning can help the model better handle complex SQL syntax, such as nested queries, joins, and business-specific calculations (e.g., calculating revenue, discounts, or aggregations). This makes fine-tuning an essential technique for improving the accuracy and reliability of text-to-SQL systems in business environments.

In summary, while general-purpose LLMs have shown promise in a variety of applications, their use in generating SQL queries for business databases requires a more specialized approach. Fine-tuning provides a way to adapt these models to the specific needs of business data, ensuring that the queries generated are both syntactically correct and contextually appropriate. This forms the basis for the experimental work presented in this thesis, where various fine-tuning strategies are explored to optimize text-to-SQL generation for business applications.

## 1.2 Research Problem

### 1.2.1 The Difficulty of Applying LLMs to Business Data

Large Language Models (LLMs), such as GPT, have shown remarkable capabilities in a wide range of natural language processing tasks. However, when it comes to applying these models to business data, especially in complex environments like relational databases, the models often face significant challenges. These difficulties arise because LLMs, which are trained on general text corpora, are not inherently equipped to handle the intricacies of business-specific data structures and operations.

In business environments, databases typically contain vast amounts of data structured into multiple interconnected tables, each with specific fields, constraints, and relationships. LLMs trained on general language data may struggle to understand the specific schema of a database, including the relationships between tables, primary and foreign keys, and the semantic meaning of terms used within the business context. For example, a model may understand the word "revenue" in general text but may fail to correctly interpret it in a specific business context where it refers to a calculation based on multiple tables and conditions.

Moreover, LLMs generally lack the ability to reason about data in the way that a database query requires. While they can generate grammatically correct natural language, generating SQL queries that interact with a complex database requires an understanding of the database's schema, business rules, and SQL syntax. This

mismatch between the model's capabilities and the task at hand is a significant barrier to applying LLMs to business data.

### 1.2.2 Specific Problems with General Models in Business Contexts

Using general-purpose models like GPT in business contexts introduces a variety of specific challenges. The most prominent of these is the inability to manage complex database schemas. Enterprise-level databases typically involve numerous tables with complex relationships. Queries often require knowledge of how these tables are interrelated, how data is aggregated, and which fields need to be referenced. General LLMs are typically trained on broad, unstructured data sources and are not inherently aware of the relational structures in databases, making it difficult for them to generate correct SQL queries.

Additionally, general-purpose models tend to generate incorrect SQL queries when faced with complex database schemas. These models are often unable to incorporate the nuances of SQL syntax, especially when handling advanced SQL constructs such as nested queries, multiple joins, or specific aggregations. For instance, while a model may generate a simple SQL query like "SELECT * FROM customers", it may struggle when asked to write a query involving a join between two tables with specific conditions.

Moreover, models that do not account for business-specific logic and rules embedded in the database schema may generate queries that are syntactically correct but semantically incorrect. For example, queries might ignore business-specific rules for data filtering, aggregation, or calculation, leading to inaccurate results that could have significant business implications.

Another issue is the lack of domain-specific knowledge in general models. In a business context, terms like "revenue", "customer", or "inventory" may have different meanings than they would in general language use. A model may understand these terms in a broad sense but fail to apply the correct interpretation in a specific business domain. For example, "revenue" in a sales database might need to account for discounts or multiple product categories, which a general model would not understand without explicit training.

### 1.2.3 The Need for Fine-Tuning Techniques

The challenges discussed above highlight the limitations of applying general-purpose LLMs to business-specific tasks like text-to-SQL generation. In order to adapt these models to the specific requirements of a business database, it is necessary to apply fine-tuning techniques that modify the model's knowledge and behavior according to the particular data and business rules involved.

Fine-tuning involves training a pre-trained model on a smaller, domain-specific dataset to adapt its behavior to a specific task. In the case of text-to-SQL generation, fine-tuning allows the model to learn the relationships within the database schema, as well as the SQL syntax and business logic that must be incorporated into the generated queries. Fine-tuning techniques enable the model to move beyond general linguistic competence and specialize in the task of generating valid SQL queries that adhere to both syntactic and semantic constraints.

However, traditional fine-tuning approaches, which involve training on a fixed dataset of query pairs (natural language to SQL), often fall short when dealing with more complex or evolving database schemas. This limitation can be overcome by using contextual fine-tuning, which adapts the model not just by providing additional examples of SQL queries but also by incorporating contextual information, such as database schema details, business rules, and syntactic rules specific to the SQL dialect in use.

Contextual fine-tuning allows the model to be conditioned with specific information at inference time, guiding it to generate SQL queries that are both syntactically correct and semantically appropriate for the business context. By incorporating domain-specific context and rules into the fine-tuning process, the model can better understand how to translate natural language queries into SQL, ensuring that the generated queries meet both the technical and business requirements of the task.

It is important to note that the fine-tuned text-to-SQL component is not intended to operate in isolation. In practical settings, it must be embedded within a larger and more complex pipeline that includes query understanding, SQL generation, execution against the database, and validation of results. This broader integration is essential to handle execution-time failures, enforce business constraints, and support robust end-to-end behavior in realistic enterprise deployments.

In conclusion, the application of general-purpose LLMs to business-specific tasks like text-to-SQL generation is hindered by a number of challenges, including difficulty in managing complex database schemas, generating correct SQL queries, and understanding domain-specific business logic. Fine-tuning techniques, particularly contextual fine-tuning, provide a means of overcoming these challenges by adapting the model to the specific needs of business databases, allowing for more accurate and reliable text-to-SQL generation.

## 1.3 Research Objectives

The primary goal of this research is to investigate the effectiveness of fine-tuning techniques, specifically Supervised Fine-Tuning (SFT) and Contextual Fine-Tuning (Context FT), in generating SQL queries from natural language questions in business contexts. This study seeks to understand how these techniques can be used to

improve the generation of SQL queries that adhere to the syntactic, semantic, and business requirements of complex enterprise databases.

### 1.3.1 Main Objective

The main objective of this research is to assess the effectiveness of Supervised Fine-Tuning (SFT) and Contextual Fine-Tuning (Context FT) for generating SQL queries from natural language input, particularly in the context of large-scale business databases. The goal is to evaluate which technique, or combination of techniques, leads to the most accurate and efficient SQL generation, while also considering the computational cost and scalability requirements that are crucial in enterprise environments.

### 1.3.2 Specific Objectives

The specific objectives of this research are as follows:

- Evaluate the effectiveness of the two fine-tuning techniques (SFT and Context FT) in improving the accuracy of SQL generation. This involves comparing the performance of models fine-tuned using these techniques on a set of complex natural language queries. The evaluation will focus on three main aspects: syntactic correctness, execution correctness, and business logic correctness. Additionally, different configurations and hybrid approaches, such as combining fine-tuning with prompting and few-shot learning, will also be tested to assess their impact on model performance.
- Analyze the trade-offs between computational cost, data efficiency, and implementation feasibility in business solutions. This objective will assess how the different fine-tuning configurations affect not only the accuracy of SQL generation but also the computational resources required (e.g., training time, inference cost, token usage). Efficiency metrics, including the number of tokens per query, will be used to analyze how each configuration affects the cost of inference and the overall feasibility of deploying these models in real-world business applications.
- Integrate the fine-tuned model into an orchestrated pipeline that better reflects a scalable use case for large enterprise databases. This objective focuses on integrating the fine-tuned models into an orchestrated pipeline, a modular system that breaks down the task of generating SQL queries into smaller, manageable sub-tasks. The orchestrator will handle query decomposition and synthesis, delegating specific sub-tasks (such as SQL generation) to specialized worker models. This design aims to create a scalable solution that can efficiently handle large datasets, as is common in enterprise environments.

In summary, the research aims to assess how fine-tuning techniques can improve SQL generation for business use cases, evaluate the efficiency of these approaches in real-world settings, and integrate the resulting models into a modular and scalable system that meets the needs of enterprise database querying.

# Chapter 2

# Background on Large Language Models and Fine-Tuning

## 2.1 Large Language Models as Foundation Models

Recent advances in artificial intelligence have led to the emergence of Large Language Models (LLMs) as a new class of general-purpose systems capable of performing a wide range of language-related tasks. Rather than being designed for a single, narrowly defined objective, modern LLMs are trained as foundation models: large-scale architectures that acquire broad linguistic, semantic, and contextual knowledge through extensive pre-training on unlabelled data. This paradigm shift has fundamentally changed how language-based systems are developed, deployed, and adapted to real-world applications.

This section provides a conceptual and architectural overview of LLMs as foundation models, with the goal of establishing the theoretical basis for the methods and system designs explored later in this thesis. It begins by defining the key properties that distinguish foundation models from traditional task-specific approaches, including transferability, emergent capabilities, and flexible adaptation mechanisms. The discussion then examines the architectural principles—most notably the Transformer and its scaling properties—that enable these models to operate effectively at unprecedented scale.

Building on this foundation, the section traces the evolution of GPT-style models, highlighting the progression from early pre-training and fine-tuning paradigms

to the emergence of in-context learning and instruction-following behavior. Particular attention is given to the transition toward more reliable, efficient, and specialized variants, culminating in the introduction of GPT-4.1. This model is of special relevance to the present work, as it represents a balance between general-purpose reasoning capabilities and practical constraints such as cost, latency, and deployability.

Finally, the section discusses how foundation models can be employed in different functional roles within larger systems, rather than as monolithic predictors. This perspective sets the stage for subsequent chapters, where Large Language Models are embedded within modular, orchestrated pipelines for complex tasks such as enterprise-scale text-to-SQL generation and data analysis.

### 2.1.1 Definition and Core Characteristics of Foundation Models

The term *foundation model* refers to a class of large-scale machine learning models trained on vast and heterogeneous datasets, typically using self-supervised or unsupervised learning objectives, and designed to serve as a general-purpose foundation for a wide range of downstream tasks. Unlike traditional task-specific models, which are trained and optimized for a single well-defined objective, foundation models acquire broad linguistic and semantic knowledge during pre-training, enabling them to be adapted to diverse applications with minimal additional supervision.

A defining property of foundation models is their reliance on large amounts of unlabelled data. By training on extensive corpora of natural language text, these models learn statistical regularities, syntactic structures, semantic relationships, and factual patterns that emerge across domains. The training objective—most commonly next-token prediction—encourages the model to internalize a general representation of language, rather than memorizing task-specific rules. As a result, the knowledge learned during pre-training can be reused across tasks without retraining the model from scratch.

Another core characteristic of foundation models is transferability. Once pre-trained, a foundation model can be adapted to new tasks through fine-tuning, prompting, or a combination of both. This contrasts with earlier machine learning paradigms, where each task required a separate model trained on a dedicated dataset. In the context of natural language processing, this transferability enables a single model to perform tasks such as text classification, summarization, question answering, code generation, and structured query generation, often with competitive performance.

Foundation models also exhibit emergent capabilities, which arise as a consequence of scale rather than explicit supervision. As the number of model parameters, training data size, and computational resources increase, new behaviors begin to

appear, including few-shot learning, in-context learning, multi-step reasoning, and implicit knowledge retrieval. These behaviors are not directly programmed into the model but instead emerge from the interaction between model architecture and large-scale training. This phenomenon distinguishes foundation models from earlier neural architectures, whose capabilities were largely bounded by their training objectives and dataset sizes.

A further distinguishing feature is the flexibility of adaptation. Foundation models can be specialized through multiple mechanisms, including supervised fine-tuning, parameter-efficient fine-tuning, and contextual adaptation via prompts. This flexibility allows practitioners to balance performance, cost, and data availability depending on the application context. In enterprise environments, where labeled data may be scarce and privacy constraints are strict, this adaptability is particularly valuable, as it enables domain-specific behavior without extensive retraining.

From a systems perspective, foundation models function not merely as standalone predictors but as general-purpose reasoning engines. Their ability to condition on complex input contexts, follow natural language instructions, and generate structured outputs allows them to be embedded within larger computational pipelines. In such systems, a foundation model may assume different roles—such as controller, planner, or executor—depending on how it is prompted and integrated. This characteristic is especially relevant for modular architectures that decompose complex tasks into smaller subtasks handled by specialized components citebommasani2021foundation.

Finally, foundation models introduce a shift in how artificial intelligence systems are designed and deployed. Rather than constructing narrowly tailored models for each task, developers can rely on a single, highly capable model as the core component of multiple applications. This paradigm reduces development overhead, simplifies maintenance, and enables rapid experimentation. However, it also raises challenges related to interpretability, reliability, and cost, which must be carefully addressed when deploying foundation models in real-world settings.

In summary, foundation models are characterized by large-scale pre-training on unlabelled data, strong transferability across tasks, emergent capabilities driven by scale, and flexible adaptation mechanisms. These properties make them uniquely suited for complex, multi-stage systems and form the conceptual basis for the Large Language Models examined in this thesis.

### 2.1.2 Transformer-Based Architectures and Scaling Principles

The rapid progress of Large Language Models as foundation models is closely tied to the adoption of the Transformer architecture and to the systematic exploration

of model scaling. Together, these two factors have enabled language models to move beyond task-specific performance and toward general-purpose reasoning and generation capabilities. Understanding both the architectural foundations and the principles governing scale is therefore essential for contextualizing the evolution of modern LLMs.

### The Transformer as the Architectural Backbone

The Transformer architecture introduced a paradigm shift in sequence modeling by replacing recurrent and convolutional structures with a fully attention-based mechanism. Instead of processing tokens sequentially, the Transformer relies on *self-attention* to compute contextualized representations, enabling each token to directly attend to all others in the input sequence.

Formally, given an input sequence represented as a matrix $X \in \mathbb{R}^{n \times d}$, where $n$ is the sequence length and $d$ the embedding dimension, self-attention is computed through three learned linear projections:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

where $Q$, $K$, and $V$ denote the *queries*, *keys*, and *values*, and $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ are trainable parameter matrices. The attention output is obtained as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

This formulation allows the model to compute a weighted combination of all token representations, where the weights reflect the relevance of each token with respect to the others. The scaling factor $\sqrt{d_k}$ stabilizes gradients when $d_k$ is large.

To increase representational capacity, the Transformer employs multi-head attention, which applies multiple attention mechanisms in parallel over different learned subspaces:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

where each head is defined as:

$$\text{head}_i = \text{Attention}(QW_Q^i, KW_K^i, VW_V^i)$$

This allows the model to capture diverse relational patterns such as syntactic dependencies, semantic roles, and positional correlations simultaneously.

Since self-attention is permutation-invariant, positional information is injected into the input embeddings using positional encodings. These can be either fixed sinusoidal functions or learned embeddings, ensuring that the model can distinguish between different token positions while preserving parallel computation.

Each Transformer layer consists of two main sublayers: a multi-head self-attention block and a position-wise feed-forward network (FFN), defined as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Residual connections and layer normalization are applied after each sublayer:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

These design choices improve gradient flow, stabilize training, and enable the construction of very deep architectures.

By removing recurrence, Transformers achieve full parallelization during training, significantly improving computational efficiency on modern hardware accelerators such as GPUs and TPUs. This scalability has been a key factor in enabling the training of large-scale language models on massive corpora, leading to the emergence of rich linguistic and semantic representations
citevaswani2017attention.

The generality and modularity of the Transformer architecture make it particularly well suited for pre-training as a foundation model and for subsequent adaptation to a wide range of downstream tasks, including structured query generation, code synthesis, and multi-step reasoning.

### Decoder-Only Transformers and Autoregressive Language Modeling

While the original Transformer architecture comprises both encoder and decoder components, most contemporary Large Language Models (LLMs) adopt a decoder-only configuration. This architectural choice is closely aligned with the autoregressive language modeling objective, which factorizes the joint probability of a token sequence $x_1, \ldots, x_n$ as:

$$P(x_1, \ldots, x_n) = \prod_{t=1}^{n} P(x_t \mid x_1, \ldots, x_{t-1})$$

The model is trained to minimize the negative log-likelihood of the next token, typically implemented as a cross-entropy loss over a large vocabulary. This formulation enables the model to learn statistical regularities in natural language through next-token prediction alone.

To enforce the causal structure required by autoregressive generation, decoder-only Transformers employ causal self-attention, also known as masked self-attention. In this setting, the attention matrix is constrained such that each token at position $t$ can attend only to positions 1 through $t$, preventing access to future tokens. Formally, this is achieved by applying a triangular mask $M$ to the attention logits:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right)V$$

where $M_{ij} = -\infty$ for $j > i$ and 0 otherwise. This ensures that the probability distribution for $x_t$ depends solely on the preceding context.

Each decoder layer consists of a masked multi-head self-attention block followed by a position-wise feed-forward network, with residual connections and layer normalization applied after each sublayer. Unlike encoder-decoder Transformers, no cross-attention mechanism is present, as the model operates entirely on a single input sequence.

The autoregressive training objective, combined with large-scale pre-training on diverse text corpora, induces the model to learn hierarchical linguistic representations. Lower layers tend to capture lexical and syntactic features, while higher layers encode semantic, pragmatic, and task-level information. Despite the simplicity of the objective, decoder-only Transformers acquire broad linguistic competence and domain-general reasoning capabilities.

A key property of decoder-only LLMs is their ability to perform in-context learning. When examples, instructions, or structured patterns are included in the input prompt, the model conditions its predictions on this context without any parameter updates. This behavior can be interpreted as implicit task adaptation through sequence modeling rather than explicit fine-tuning.

From a functional perspective, decoder-only Transformers support a wide range of behaviors, including:

- Natural language understanding and generation
- Instruction following
- Code and structured query synthesis
- Multi-step reasoning and planning

In the context of structured generation tasks such as Text-to-SQL, the decoder-only architecture is particularly effective because it treats SQL tokens as part of the same autoregressive sequence as the natural language input. The model learns to map linguistic patterns to syntactic SQL constructs, while respecting grammar constraints implicitly learned from data.

However, the autoregressive formulation also introduces limitations. Errors made early in the generation process can propagate through the remainder of the sequence, leading to compounding mistakes. Moreover, the model lacks an explicit mechanism for global planning or verification, relying instead on token-level probabilities.

Despite these limitations, decoder-only Transformers remain the dominant architecture for large-scale language models due to their simplicity, scalability, and versatility. Their capacity to unify language understanding, generation, and structured output within a single probabilistic framework makes them a foundational component of modern AI systems. [1]

### From Transformers to Large Language Models

The transition from the original Transformer architecture to modern Large Language Models (LLMs) was primarily driven by systematic scaling in model size, training data, and computational resources. While the Transformer was initially proposed for machine translation, its general-purpose design made it suitable for a wide range of sequence modeling tasks.

Early Transformer-based models operated at relatively small scales, with tens or hundreds of millions of parameters. However, empirical studies demonstrated that increasing model capacity and dataset size leads to predictable and consistent improvements in performance across diverse tasks. This observation gave rise to the concept of scaling laws, which describe how model loss decreases as a function of parameter count, dataset size, and compute budget.

Formally, the training loss $L$ of a language model follows approximate power-law relationships of the form:

$$L(N, D, C) \approx aN^{-\alpha} + bD^{-\beta} + cC^{-\gamma}$$

where $N$ is the number of model parameters, $D$ is the size of the training dataset, and $C$ represents the compute budget. These scaling laws provided a theoretical and empirical justification for training increasingly large Transformer models.

As models grew to billions or trillions of parameters, they began to exhibit emergent capabilities—qualitative behavioral changes that were not present at smaller scales. These include:

- Instruction following without task-specific fine-tuning
- Few-shot and zero-shot learning
- Code and structured query generation
- Multi-step reasoning and tool usage

Such capabilities arise from the interaction between the autoregressive training objective, the self-attention architecture, and exposure to large, diverse corpora. Importantly, these behaviors are not explicitly programmed, but emerge as a consequence of scale.

Modern LLMs typically consist of dozens to hundreds of Transformer decoder layers, with hidden dimensions and attention heads significantly larger than those used in earlier models. The depth of these networks enables the formation of hierarchical representations, where lower layers capture lexical and syntactic features, while higher layers encode semantic, pragmatic, and task-level abstractions.

In addition to architectural scaling, training pipelines for LLMs incorporate advanced optimization techniques, such as adaptive learning rate schedules, large-batch training, mixed-precision computation, and distributed model parallelism. These engineering advances make it feasible to train models on trillion-token datasets.

The result is a class of models that function as foundation models: general-purpose systems trained on broad data that can be adapted to many downstream tasks through prompting or fine-tuning. Rather than being specialized for a single application, LLMs serve as flexible computational substrates for tasks such as natural language understanding, code synthesis, and structured query generation.

This evolution from task-specific Transformers to large-scale foundation models underpins the design of modern AI systems, where a single pretrained model can be repurposed for multiple roles, including orchestration, reasoning, and execution within modular pipelines. [1]

### Scaling Laws and Emergent Behavior

One of the most significant discoveries in the development of LLMs is the existence of predictable relationships between model scale and performance. Empirical studies have shown that increasing the number of parameters, the size of the training dataset, and the amount of compute leads to smooth and consistent improvements across a wide range of tasks. These relationships, often referred to as scaling laws, suggest that many capabilities of LLMs emerge gradually as a function of scale rather than as a result of explicit architectural changes.

As models grow larger, they begin to exhibit emergent behaviors that are absent or weak in smaller models. These include few-shot learning, where the model can perform new tasks based on a small number of examples provided at inference time, as well as improved reasoning, abstraction, and contextual understanding. Importantly, these capabilities are not directly supervised during training but arise from the interaction between large model capacity and diverse training data.

The recognition of scaling laws has had a profound impact on research directions, motivating the development of increasingly large models and more efficient training strategies. At the same time, it has shifted the focus from designing task-specific architectures to optimizing general-purpose models that can adapt to many tasks through prompting or fine-tuning.

### Compute-Optimal Training and Data Scaling

While early scaling efforts emphasized increasing parameter counts, later research highlighted the importance of balancing model size with dataset size and training duration. It was observed that many large models were undertrained relative to their capacity, leading to suboptimal performance. This insight gave rise to the concept of compute-optimal training, which aims to allocate computational resources more effectively by adjusting both model size and the number of training tokens.

Compute-optimal scaling demonstrates that, given a fixed computational budget, training moderately sized models on larger and more diverse datasets can outperform training extremely large models on limited data. This shift has reinforced the importance of data quality, token diversity, and training efficiency in the design of modern LLMs.

These principles are particularly relevant in practical settings, where computational resources are finite and cost considerations play a central role. As a result, recent LLM development has increasingly focused on efficiency-aware scaling rather than unbounded growth in parameter counts.

### Implications for Foundation Models

The combination of Transformer-based architectures and well-understood scaling principles underpins the success of LLMs as foundation models. The ability to scale both model capacity and data exposure allows these systems to acquire general-purpose knowledge that can later be adapted to specialized tasks. Furthermore, the architectural simplicity and flexibility of the Transformer facilitate reuse across different domains and applications.

From a system design perspective, these properties enable the use of a single model as a shared backbone for multiple functionalities, ranging from high-level reasoning and orchestration to low-level execution tasks. This insight is central to the design choices explored later in this thesis, where different configurations and adaptations of the same foundation model are employed within a modular, multi-stage pipeline.

In summary, Transformer-based architectures provide the structural foundation that makes large-scale language modeling feasible, while scaling principles explain how increased capacity and data exposure lead to improved and emergent capabilities. Together, these factors have enabled the rise of Large Language Models as versatile foundation models, capable of supporting complex reasoning and domain-specific adaptation in real-world systems.

### 2.1.3 The Evolution of GPT-Style Models

The Generative Pre-trained Transformer (GPT) family represents one of the most influential lines of development in the history of Large Language Models. GPT-style models are characterized by a decoder-only Transformer architecture trained using an autoregressive language modeling objective. Their evolution illustrates a progressive shift from task-specific fine-tuning toward general-purpose reasoning, instruction following, and in-context learning, driven primarily by increases in scale, data diversity, and training sophistication.

#### GPT-1: Pre-Training as a Universal Initialization

The first GPT model introduced the now-standard paradigm of large-scale pre-training followed by task-specific adaptation. GPT-1 demonstrated that training a Transformer decoder on a large corpus of unlabelled text using a next-token prediction objective could yield a model that captures broad linguistic and semantic knowledge [2]. Rather than optimizing directly for downstream tasks, the model learns a general representation of language that can later be adapted with relatively small amounts of supervised data.

This approach marked a departure from earlier neural models, which were typically trained independently for each task. GPT-1 showed that a single pre-trained model could serve as a universal initialization for diverse applications, including text classification, question answering, and summarization. Although limited in size by modern standards, GPT-1 validated the foundation model paradigm and laid the conceptual groundwork for subsequent scaling efforts.

#### GPT-2: Scaling and Emergent Generative Capabilities

The release of GPT-2 represented a significant increase in model size and training data, leading to substantial qualitative improvements in text generation. GPT-2 exhibited a remarkable ability to generate long, coherent passages of text that maintained contextual consistency across multiple paragraphs. This behavior emerged without explicit supervision for discourse-level coherence, highlighting the role of scale in unlocking new capabilities.

GPT-2 also revealed that large language models could perform a variety of tasks implicitly through prompting, even without fine-tuning. For example, by framing inputs appropriately, the model could perform translation, summarization, or question answering in a zero-shot setting. These observations provided early evidence that language models could internalize task-relevant patterns during pre-training, reducing the need for explicit task-specific training.

### GPT-3: In-Context Learning and Few-Shot Generalization

GPT-3 marked a major inflection point in the evolution of GPT-style models. With a parameter count orders of magnitude larger than its predecessors, GPT-3 demonstrated that scale alone could enable models to perform new tasks through in-context learning. Instead of updating model parameters via fine-tuning, GPT-3 could infer the desired task behavior from a small number of examples embedded directly in the input prompt.

This capability enabled three distinct modes of operation:

- Zero-shot learning, where the model performs a task based solely on a natural language instruction.
- One-shot learning, where a single example is provided.
- Few-shot learning, where a small set of examples defines the task.

In-context learning fundamentally changed how practitioners interacted with language models. Tasks could be specified dynamically at inference time, without retraining or modifying model parameters. This flexibility significantly reduced development overhead and accelerated experimentation, particularly in applied settings where labeled data is scarce or expensive to obtain.

However, GPT-3 also exposed limitations, including sensitivity to prompt phrasing, inconsistent reasoning behavior, and a tendency to produce confident but incorrect outputs. These shortcomings highlighted the need for improved alignment, robustness, and reasoning capabilities.

### GPT-4: Reliability, Reasoning, and Multimodal Capabilities

GPT-4 represents a further evolution of the GPT paradigm, emphasizing improved reliability, stronger reasoning abilities, and broader applicability. Compared to earlier models, GPT-4 exhibits more consistent instruction following, better handling of complex multi-step tasks, and improved robustness to ambiguous or underspecified inputs. These improvements reflect advances not only in scale, but also in training methodology, data curation, and alignment techniques.

A key distinction of GPT-4 is its enhanced reasoning performance. The model demonstrates stronger capabilities in tasks that require logical inference, structured problem decomposition, and the integration of multiple pieces of information. This makes GPT-4 particularly well suited for applications involving analytical reasoning, code generation, and structured data interaction.

GPT-4 also introduced multimodal capabilities, enabling the model to process and reason over both textual and visual inputs. This extension reinforces the role of GPT-style models as general-purpose cognitive systems rather than purely text-based generators. Although multimodality is not the primary focus of this thesis,

it underscores the broader trajectory of GPT models toward unified, multi-domain intelligence.

#### From Monolithic Models to System Components

As GPT-style models evolved, their role shifted from standalone predictors to core components within larger systems. Rather than relying on a single model invocation to solve complex problems, practitioners increasingly embed GPT models within multi-step pipelines that incorporate decomposition, tool use, error handling, and synthesis. This shift reflects an understanding that even highly capable models benefit from structured interaction patterns and external constraints.

The evolution from GPT-1 to GPT-4 thus illustrates not only improvements in raw model capability, but also a broader change in how language models are used. Modern GPT-style models function as adaptable building blocks that can assume different roles—such as planner, executor, or analyst—depending on how they are prompted and integrated into a system.

#### Summary

The progression of GPT-style models demonstrates how scaling, architectural simplicity, and large-scale pre-training have transformed language models into powerful foundation systems. From GPT-1's introduction of universal pre-training, through GPT-3's in-context learning, to GPT-4's emphasis on reasoning and reliability, each generation has expanded the scope of what language models can achieve. This evolution sets the stage for the discussion of specialized variants such as GPT-4.1, which aim to balance general-purpose capability with efficiency and deployability in real-world applications.

### 2.1.4 GPT-4.1 and Model Specialization

The introduction of GPT-4.1 marks an important step in the evolution of Large Language Models from general-purpose research systems to deployable components of real-world applications
citeopenai2025gpt41. While GPT-4 demonstrated state-of-the-art performance across a wide range of complex reasoning and language understanding tasks, its high computational cost, memory footprint, and inference latency limit its practicality in production environments.

GPT-4.1 represents a specialized variant within the GPT-4 model family, designed to preserve core reasoning and instruction-following capabilities while improving efficiency, cost-effectiveness, and scalability. Rather than introducing a

fundamentally new architecture, GPT-4.1 reflects a shift toward model specialization, where performance is optimized along multiple operational dimensions, including inference speed, resource consumption, and deployment reliability.

This transition illustrates a broader trend in the foundation model paradigm: as models mature, emphasis moves from maximizing raw capability to optimizing the trade-offs between expressiveness, robustness, latency, and economic feasibility.

### Motivation for GPT-4.1

As LLMs scale in size and capability, their deployment becomes increasingly constrained by practical system-level considerations. Large models incur substantial inference costs due to high parameter counts, large activation tensors, and extensive attention computations. These factors result in increased latency, higher memory requirements, and greater energy consumption.

In enterprise and interactive settings, systems must support:

- concurrent user requests,
- low-latency responses,
- predictable operational costs,
- sustained availability over long time horizons.

GPT-4.1 was introduced to address these constraints by offering a model that retains the reasoning and instruction-following strengths of GPT-4 while reducing computational overhead. This enables its use in scenarios where consistent performance and scalability are more critical than maximal reasoning depth.

Rather than replacing GPT-4, GPT-4.1 occupies a different point in the design space, balancing capability and efficiency to better align with real-world deployment requirements.

### Architectural and Training Considerations

Although the internal architectural details of GPT-4.1 are not publicly disclosed, it can be characterized as a Transformer-based, decoder-only language model consistent with the GPT-4 lineage. The specialization of GPT-4.1 arises not from a radical architectural departure, but from a combination of factors including controlled parameter scaling, optimized training procedures, curated datasets, and alignment-focused fine-tuning.

Key factors contributing to GPT-4.1's specialization include:

- Parameter scaling constraints, limiting model size to reduce memory usage and inference cost;

- Optimized training pipelines, including improved data filtering and curriculum strategies;
- Alignment fine-tuning, ensuring reliable instruction following and reduced output variance;
- Inference optimization, enabling faster decoding and lower latency.

By constraining model capacity and optimizing training for inference efficiency, GPT-4.1 achieves lower memory usage and faster response times compared to GPT-4, making it more suitable for deployment in high-frequency, interactive, and multi-step reasoning systems.

### GPT-4.1 Mini: Efficiency-Oriented Specialization

Within the GPT-4.1 family, the Mini variant represents a further step toward efficiency-oriented specialization. GPT-4.1 Mini is designed to operate under tighter computational budgets while maintaining a strong baseline of linguistic competence and reasoning ability.

Although it does not match the full GPT-4 model in terms of deep reasoning, long-context understanding, or complex abstraction, GPT-4.1 Mini offers a favorable balance between performance and operational cost. Its design prioritizes:

- low inference latency,
- reduced memory footprint,
- stable instruction-following behavior,
- robustness across structured generation tasks.

These properties make GPT-4.1 Mini particularly suitable for systems that require frequent model invocations, predictable runtime behavior, and scalable deployment. In such contexts, overall system performance depends not only on the strength of individual model calls, but on the efficiency and reliability of the entire processing pipeline.

### Specialization by Role: Reasoning vs Execution

A central implication of GPT-4.1's design is the recognition that different stages of a complex pipeline benefit from different model characteristics. High-level reasoning, planning, and synthesis require models that excel at abstraction, instruction following, and contextual integration. In contrast, execution-oriented tasks benefit from consistency, precision, and efficiency.

GPT-4.1 Mini is particularly well suited for the former role. Its ability to decompose complex user queries, reason over partial results, and synthesize structured outputs makes it an effective orchestrator model. Conversely, more narrowly fine-tuned models—such as those optimized for SQL generation—can be employed as worker models focused on execution-level subtasks.

This separation of concerns reflects a broader shift from monolithic model usage to role-based specialization within LLM systems. Rather than relying on a single model to handle all aspects of a task, modern architectures distribute responsibilities across multiple models that are specialized through fine-tuning, prompting, or architectural constraints.

Such modular designs improve system robustness, reduce computational overhead, and enable scalable multi-step reasoning pipelines, as demonstrated in agent-based orchestration frameworks.

### Implications for Domain Adaptation

GPT-4.1 also plays a crucial role in enabling effective domain adaptation strategies. Because it serves as a strong general-purpose reasoning model, it can coordinate and contextualize the outputs of domain-specialized components. In the context of this thesis, GPT-4.1 Mini functions as the central reasoning engine that interfaces with a fine-tuned SQL generation model.

This design allows domain adaptation to occur at multiple levels:

- at the model level, through supervised fine-tuning of worker models,
- at the context level, through schema-aware prompting and few-shot examples,
- at the system level, through orchestration and decomposition strategies.

By decoupling reasoning from execution, GPT-4.1 enables more robust and maintainable systems, where changes to the domain or database schema can often be addressed by updating a single component rather than retraining the entire model stack.

### Practical Considerations and Trade-Offs

While GPT-4.1 Mini offers substantial advantages in efficiency and scalability, it also introduces trade-offs. Compared to larger models, it may exhibit reduced performance on tasks that require deep logical inference, long-context retention, or highly nuanced language understanding. These limitations must be accounted for when designing systems that rely on GPT-4.1 as a core component.

However, in many applied scenarios, these limitations are offset by gains at the system level. Lower latency enables more interactive user experiences, reduced

costs allow broader deployment, and predictable performance simplifies system engineering. When combined with decomposition strategies and specialized worker models, GPT-4.1 Mini can support complex analytical workflows that would be impractical with a single large model invocation.

**Summary**

GPT-4.1 exemplifies the transition from monolithic, capability-maximizing language models to specialized, efficiency-oriented foundation models designed for real-world deployment. By balancing reasoning ability with practical constraints, GPT-4.1—and in particular its Mini variant—provides a strong foundation for modular, orchestrated systems. This specialization enables Large Language Models to be integrated effectively into complex pipelines, setting the stage for the system architectures and experimental evaluations explored in subsequent chapters of this thesis.

### 2.1.5 General-Purpose Models vs Task-Oriented Usage

One of the most significant conceptual shifts introduced by Large Language Models as foundation models is the distinction between a model's general-purpose capability and its task-oriented usage within a system. While modern LLMs are trained to perform a wide range of tasks using a single set of parameters, their effectiveness in real-world applications depends heavily on how they are employed, specialized, and integrated into larger computational workflows.

General-purpose language models, such as GPT-style architectures, are designed to acquire broad linguistic competence through large-scale pre-training. This enables them to understand natural language instructions, generate coherent text, reason over abstract concepts, and adapt to new tasks with minimal supervision. However, this versatility does not imply that a single model invocation is optimal for all tasks, particularly in complex or high-stakes environments such as enterprise data analysis.

In practice, different tasks impose different requirements on a language model. High-level reasoning tasks—such as interpreting user intent, decomposing complex questions, or synthesizing insights across multiple sources—benefit from models that excel at abstraction, instruction following, and contextual integration. In contrast, execution-oriented tasks—such as generating syntactically correct SQL queries or interacting with structured systems—require precision, consistency, and strict adherence to formal constraints. Attempting to address both categories of tasks with a single, monolithic model invocation often leads to suboptimal results.

This observation motivates a shift from treating LLMs as standalone predictors to viewing them as configurable components whose behavior depends on their role

within a system. A general-purpose model can be adapted to different roles through prompting, fine-tuning, or architectural constraints, effectively transforming the same underlying model into a planner, controller, or executor. This role-based usage allows system designers to leverage the strengths of foundation models while mitigating their weaknesses.

Task-oriented usage further benefits from explicit specialization. Rather than relying exclusively on a general-purpose model, systems can incorporate task-specific models that are fine-tuned or otherwise optimized for narrow objectives. In such architectures, a general-purpose LLM may be responsible for high-level reasoning and coordination, while specialized models handle well-defined subtasks. This separation of concerns improves reliability, interpretability, and maintainability, especially in domains where correctness is critical.

From a system design perspective, the distinction between general-purpose and task-oriented usage enables modularity. Components can be developed, evaluated, and updated independently, reducing the risk that changes in one part of the system will negatively impact overall behavior. For example, improvements to a task-specific SQL generation model do not require retraining the general-purpose reasoning model, and vice versa. This modular approach aligns well with enterprise requirements for scalability and long-term maintenance.

Furthermore, task-oriented usage has important implications for cost and efficiency. Large general-purpose models are often expensive to invoke repeatedly, particularly in workflows that involve multiple intermediate steps. By reserving such models for tasks that genuinely require their capabilities and delegating simpler or more constrained tasks to specialized models, systems can achieve better overall performance at lower cost. This consideration is especially relevant in pipelines that process large volumes of data or support many concurrent users.

In summary, while foundation models provide a powerful general-purpose substrate for language understanding and generation, their effective use in real-world systems requires careful consideration of task-specific requirements. Distinguishing between general-purpose reasoning and task-oriented execution enables the design of modular, efficient, and robust architectures. This perspective forms the conceptual basis for the orchestrator-worker paradigm explored later in this thesis, where different models are employed in complementary roles to address complex analytical tasks.

### 2.1.6 Implications for Enterprise Systems

The emergence of Large Language Models as foundation models has profound implications for enterprise systems, particularly in domains that require scalable data access, reliability, and cost-aware deployment. While the general-purpose

capabilities of modern LLMs enable unprecedented flexibility, their effective integration into enterprise environments requires careful consideration of practical constraints that extend beyond raw model performance.

One of the primary challenges in enterprise adoption is scalability. Enterprise systems often serve multiple users concurrently and must process large volumes of data with predictable latency. Invoking a large, general-purpose language model for every step of a complex workflow can quickly become computationally expensive and operationally inefficient. As a result, enterprises must design systems that leverage the strengths of foundation models while minimizing unnecessary model usage. This requirement reinforces the importance of role-based specialization, where high-capability models are reserved for tasks that require reasoning and synthesis, and more constrained models are employed for execution-oriented subtasks.

Cost efficiency represents another critical concern. Unlike traditional software components, LLMs incur usage-based costs that scale with both input size and invocation frequency. In analytical workflows that involve iterative reasoning or large datasets, naïvely applying an LLM to raw data can lead to prohibitive costs. Enterprise-grade systems therefore favor architectures that reduce token consumption by operating on aggregated or summarized data, delegating heavy computation to traditional databases and using LLMs primarily for interpretation and decision-making. This approach aligns naturally with decomposition-based pipelines, where each model invocation produces compact, information-dense outputs.

Data privacy and governance further shape how foundation models are deployed in enterprise contexts. Proprietary databases often contain sensitive information that cannot be exposed indiscriminately to external systems. This constraint limits the feasibility of large-scale supervised fine-tuning on raw enterprise data and encourages approaches that minimize data exposure. Contextual adaptation, parameter-efficient fine-tuning, and modular system design allow organizations to incorporate domain knowledge while maintaining control over data access and compliance requirements.

Reliability and correctness are also paramount in enterprise settings. Unlike consumer-facing applications, analytical and decision-support systems must produce outputs that are interpretable, verifiable, and reproducible. Foundation models, while powerful, may generate incorrect or inconsistent responses if used without constraints. Integrating LLMs into structured pipelines—with explicit task boundaries, validation mechanisms, and error-handling strategies—improves robustness and makes system behavior more predictable. This structured integration contrasts with monolithic, single-shot model usage, which often lacks transparency and control.

From a system engineering perspective, foundation models enable a shift toward modular architectures. Rather than embedding intelligence directly into each application component, enterprises can rely on a shared language model backbone that supports multiple use cases. This modularity simplifies maintenance, facilitates

iterative improvement, and allows individual components—such as task-specific models or prompting strategies—to evolve independently. As business requirements change, systems can be adapted incrementally without requiring full retraining or redesign.

Finally, the implications of foundation models extend to how enterprise workflows are conceptualized. Instead of treating natural language interfaces as simple query translators, organizations can design multi-stage analytical processes where LLMs assist with problem decomposition, hypothesis generation, and result interpretation. In such workflows, the role of the language model shifts from passive responder to active participant in the analytical process, coordinating traditional computation with human-like reasoning.

In summary, while Large Language Models offer powerful general-purpose capabilities, their successful deployment in enterprise systems depends on architectural choices that address scalability, cost, privacy, and reliability. Foundation models are most effective when embedded within modular, role-aware pipelines that combine classical data processing with language-based reasoning. These considerations provide the practical motivation for the fine-tuning strategies and orchestrated system architectures explored in the subsequent sections of this thesis.

## 2.2 Fine-Tuning Strategies for Domain Adaptation

While Large Language Models acquire broad linguistic and semantic knowledge during pre-training, their direct application to domain-specific tasks often leads to suboptimal performance. This limitation is particularly evident in enterprise settings, where models must operate over structured data, adhere to strict semantic constraints, and reflect domain-specific business logic. Fine-tuning strategies play a crucial role in bridging the gap between general-purpose language understanding and specialized task requirements [3, 4].

This section reviews the principal approaches used to adapt foundation models to specific domains and tasks. Rather than viewing fine-tuning as a single technique, it is presented here as a spectrum of strategies that vary in terms of supervision, data requirements, computational cost, and flexibility. The discussion begins with Supervised Fine-Tuning (SFT), the most direct and widely used method for domain adaptation, before moving on in subsequent sections to contextual and parameter-efficient alternatives.

The goal of this section is to provide the theoretical foundation necessary to understand the design choices made in this thesis, particularly the comparison between supervised fine-tuning and context-based adaptation for text-to-SQL generation in enterprise environments.

### 2.2.1 Supervised Fine-Tuning (SFT)

Supervised Fine-Tuning (SFT) is one of the most established and effective methods for adapting Large Language Models to domain-specific tasks [3, 4]. In SFT, a pre-trained language model is further trained on a labeled dataset that directly reflects the target task. For text-to-SQL generation, this typically consists of pairs of natural language questions and their corresponding SQL queries, often augmented with schema information or execution context.

The core objective of SFT is to align the model's output distribution with the desired task behavior. By exposing the model to examples that explicitly demonstrate correct input-output mappings, SFT encourages the model to internalize task-specific patterns that may not be sufficiently represented in its original pre-training data. This process allows the model to move beyond general linguistic competence and develop a more precise understanding of domain semantics and structural constraints.

#### Training Procedure

The supervised fine-tuning process generally follows a standard pipeline. A pre-trained foundation model serves as the initialization point, leveraging its broad knowledge of language and syntax. The model is then trained on a curated dataset using a supervised learning objective, typically minimizing the negative log-likelihood of the target output sequence given the input.

In the case of text-to-SQL tasks, the input often includes:

- a natural language question,
- a description of the database schema,
- optional contextual information such as table relationships or constraints.

The target output is a syntactically valid and semantically correct SQL query. During training, the model learns to map linguistic intent to structured query logic, including table selection, join conditions, filtering predicates, aggregation functions, and grouping clauses.

#### Advantages of Supervised Fine-Tuning

One of the primary advantages of SFT is its ability to achieve high task-specific accuracy. By directly optimizing for the target output, the model becomes highly specialized and reliable within the scope of the training data. This is particularly valuable in structured tasks such as SQL generation, where even minor errors can lead to incorrect or non-executable queries.

SFT also offers a degree of predictability and consistency that is often lacking in purely prompt-based approaches. Once fine-tuned, the model's behavior is less sensitive to prompt phrasing and more robust across different input formulations. This stability is especially important in enterprise environments, where reproducibility and correctness are critical.

Another key benefit of SFT is its compatibility with downstream system integration. A fine-tuned model can be treated as a task-specific component within a larger pipeline, with well-defined input and output behavior. This makes it suitable for use as an execution-oriented model in modular architectures, such as worker components in orchestrato-worker systems.

### 2.2.2 Parametric Knowledge, Knowledge Editing, and Resistance to Rule Updates

A recurring phenomenon observed in practice is that a fine-tuned language model may rapidly internalize new, previously unseen factual information, such as the structure of a proprietary database schema or the names of domain-specific tables and attributes, while simultaneously failing to reliably adopt rule-level constraints that conflict with generation patterns acquired during large-scale pre-training. In the context of Text-to-SQL systems, this discrepancy is particularly evident when enforcing dialect-specific constraints, such as MySQL 8 strict modes (e.g., `ONLY_FULL_GROUP_BY`), which frequently contradict permissive heuristics implicitly learned from heterogeneous SQL corpora.

This behavior should not be interpreted as the model "forgetting" or ignoring newly provided rules. Rather, it reflects a form of resistance to parameter-level updates when the desired behavior competes with highly consolidated priors encoded in the model's weights. From a representational perspective, factual knowledge (e.g., table names, foreign key relationships) tends to be localized and additive, whereas procedural constraints and execution semantics are distributed across many parameters and strongly entangled with general language and code generation patterns.

In the literature, this phenomenon is closely related to the broader problem of knowledge editing, also referred to as model editing. Knowledge editing studies investigate how—and under what conditions—a pre-trained model can be modified to change specific facts, behaviors, or constraints encoded in its parametric memory [5]. Methods such as ROME demonstrate that even seemingly localized updates often require carefully targeted interventions to reliably alter internal representations without inducing unintended side effects on unrelated behaviors [6]. Subsequent work extends model editing to larger sets of simultaneous updates (e.g., MEMIT), further highlighting the practical challenges of maintaining locality and minimizing side effects [7, 8]. These findings suggest that modifying procedural or rule-based

knowledge can be substantially more challenging than injecting new declarative information.

From the perspective of supervised fine-tuning, this resistance can also be framed as a competing-objective problem. When the fine-tuning dataset is relatively small and the target behavior contradicts high-frequency patterns learned during pre-training, gradient updates may be insufficient to shift the model toward the desired constraint manifold. In the case of SQL generation, permissive grouping strategies, loose alias usage, or dialect-agnostic window function patterns may dominate the learned distribution. Regularization effects, stability constraints, and optimization dynamics that favor the preservation of previously learned capabilities further limit the effective magnitude of these updates. This phenomenon aligns with well-studied issues in continual learning, where models are encouraged to incorporate new knowledge while avoiding catastrophic forgetting of prior skills [9].

Crucially, this framing helps explain why in-context guidance often proves more effective than limited supervised fine-tuning for enforcing strict, dialect-specific constraints. In-context learning operates at inference time, directly influencing the decoding process by injecting explicit constraints and, more importantly, procedural demonstrations of correct behavior. Rather than attempting to overwrite parametric priors, in-context examples condition the model's next-token distribution dynamically, enabling them to dominate entrenched generation heuristics when properly constructed.

Recent empirical work on in-context learning supports this interpretation, suggesting that demonstrations act as strong behavioral conditioning signals rather than mere reminders of factual information [10]. In procedural tasks such as SQL generation, demonstrations encode not only what the correct output looks like, but how intermediate reasoning steps should be structured, including ordering of aggregations, use of CTEs, and separation between aggregation and windowing stages.

Taken together, these observations point to a fundamental distinction between two forms of specialization in Text-to-SQL systems. Schema specialization primarily involves the acquisition of new relational knowledge and can be effectively achieved through supervised fine-tuning. In contrast, dialect-compliance specialization often requires overriding deeply ingrained procedural priors and is therefore more amenable to inference-time control mechanisms, such as structured prompts and few-shot exemplars.

This distinction directly motivates the hybrid strategies evaluated in this thesis. Supervised fine-tuning is leveraged to internalize schema-specific knowledge and common query templates, while prompt-level rules and few-shot demonstrations are employed to reliably enforce MySQL 8 execution constraints. As the experimental results in Chapter 3 will show, this separation of responsibilities aligns more naturally with the adaptation dynamics of large language models than attempting

to encode all constraints directly into the model parameters.

### Limitations and Challenges

Despite its effectiveness, supervised fine-tuning introduces several challenges that become particularly salient in settings where the target behavior competes with strong pre-trained priors. The most significant limitation is the requirement for labeled training data. Constructing high-quality datasets of natural language queries and corresponding SQL statements is time-consuming and often requires domain expertise; moreover, producing targeted supervision for dialect-specific edge cases (e.g., strict grouping and windowing patterns under MySQL 8) can be substantially more expensive than collecting generic schema-coverage examples. In enterprise contexts, data availability may be further constrained by privacy, confidentiality, or regulatory considerations.

SFT also carries the risk of overfitting to the training domain. When the dataset is small, models may memorize surface patterns without reliably learning the underlying procedural constraints, leading to brittle behavior on unseen query formulations or higher-complexity compositions. This rigidity is exacerbated when databases evolve (schema drift) or when query distributions change over time, reducing the long-term maintainability of purely SFT-based solutions.

From a computational perspective, supervised fine-tuning incurs additional training costs and infrastructure requirements. While these costs are typically lower than those of full pre-training, they may still be non-trivial for frontier models and may increase rapidly as the dataset is expanded to overcome knowledge-editing resistance. Moreover, repeated fine-tuning cycles may be necessary as domain requirements evolve, increasing operational complexity.

### SFT in Enterprise Text-to-SQL Applications

In the context of enterprise text-to-SQL generation, SFT remains a powerful adaptation strategy when high accuracy and execution correctness are prioritized. By exposing the model to representative examples of business-specific queries, SFT enables precise alignment with domain semantics, table relationships, and naming conventions.

However, the limitations of SFT motivate the exploration of complementary approaches. In practice, SFT is often combined with few-shot prompting, schema-aware context injection, or modular system design to mitigate data scarcity and improve generalization. These hybrid strategies allow organizations to benefit from the strengths of supervised fine-tuning while maintaining flexibility and scalability.

In this thesis, SFT is evaluated not as an isolated technique, but as a component within a broader system architecture. Its performance is assessed both in isolation and as part of a decomposed, orchestrated pipeline, providing insight into how

supervised adaptation interacts with higher-level reasoning and system-level design choices.

### 2.2.3 Contextual Fine-Tuning and In-Context Learning

While Supervised Fine-Tuning (SFT) directly adapts a model to a specific task by training it on labeled data, Contextual Fine-Tuning (i.e., prompt-based adaptation via in-context learning) represents a more flexible and data-efficient approach [1, 11]. In this paradigm, a model is conditioned to adapt to a specific task not through retraining of its parameters, but through the inclusion of task-specific context, examples, and instructions provided during inference. This allows the model to generate task-specific outputs without significant changes to the underlying architecture, making it an attractive option in resource-constrained environments or when domain-specific labeled data is limited.

Contextual fine-tuning is particularly effective when combined with few-shot and zero-shot settings, where the model generalizes to new tasks from a small number of demonstrations and/or high-level instructions [1, 10]. Unlike SFT, which requires parameter updates on a labeled dataset, contextual adaptation leverages the model's pre-trained knowledge and steers generation through the prompt.

#### Concept of Contextual Fine-Tuning

Contextual fine-tuning involves providing the model with a prompt that contains relevant task information, schema details, and sometimes example inputs and outputs. The model is then able to generate a response based on the patterns it has learned during pre-training and adapt it to the current task.

This technique can be viewed as a form of "soft adaptation," where the model remains largely unchanged but is guided to produce task-specific behavior through the prompt structure. The key advantage of this approach is its ability to adapt to a wide range of tasks without requiring retraining, making it suitable for tasks with limited labeled data or rapidly changing domains.

In practice, contextual fine-tuning typically involves the following components:

- Task Description: A clear instruction that specifies what the model is expected to do, such as "Generate an SQL query from the following natural language question."
- Contextual Information: This may include relevant background information, schema definitions, or business logic that the model must consider while generating the output.
- Examples (Few-shot or Zero-shot): The model is provided with a few examples of input-output pairs that demonstrate the task structure. Few-shot examples

are particularly useful for tasks with limited training data, while zero-shot can be used when no task-specific data is available.

By conditioning the model with the right context, it becomes capable of generalizing to new tasks while leveraging its pre-trained linguistic and factual knowledge. This allows for the generation of coherent and semantically relevant outputs, even when the model has never explicitly seen a particular task during pre-training.

### Few-Shot and Zero-Shot Learning in Contextual Fine-Tuning

Contextual fine-tuning can be used in conjunction with few-shot and zero-shot learning, two powerful techniques that allow the model to perform tasks with minimal task-specific data. These methods are particularly effective in scenarios where labeled training data is scarce, expensive, or unavailable.

- Zero-shot learning refers to the model's ability to generalize to a task with no direct examples provided during inference. Instead, the model relies entirely on the task description and its pre-trained knowledge. For example, a natural language instruction such as "Translate the following text into French" allows the model to perform translation without ever having seen any training examples of text-to-text translation [1].
- Few-shot learning extends zero-shot learning by providing a small number of input-output examples in the prompt, typically ranging from one to five examples. These examples act as a template, showing the model how to map specific inputs to desired outputs. Few-shot learning has proven highly effective in tasks like text generation, question answering, and structured query generation, allowing models to quickly adapt to new tasks with minimal supervision [1, 10].

Both techniques rely heavily on the model's ability to extract patterns from the prompt and use them for task-specific inference, demonstrating the power of contextual adaptation to guide the model's behavior in real-time [10].

### Applications of Contextual Fine-Tuning in SQL Generation

Contextual fine-tuning has significant advantages in tasks like text-to-SQL generation, where the objective is to translate natural language questions into valid SQL queries. In such cases, the model's ability to reason over structured data, apply domain knowledge, and adhere to syntactic constraints makes contextual fine-tuning especially valuable.

In this application, the process typically follows these steps:

- The model receives a natural language question (e.g., "What is the total sales by region for the last quarter?") as the input.
- The model is provided with schema information about the relevant tables (e.g., tables for sales, regions, and time).
- Few-shot examples are included in the prompt to show how questions are typically translated into SQL queries (e.g., "Show me all customers in region X" → "SELECT * FROM customers WHERE region = 'X'").
- The model then generates an SQL query that reflects the structure of the given question while considering the schema and the examples provided.

This process enables the model to handle diverse and complex SQL generation tasks without needing to be fine-tuned on a large dataset of queries. The flexibility of contextual fine-tuning allows for the generation of queries that are tailored to specific user requirements, adapting to new databases or schema changes with minimal intervention.

### Advantages and Limitations of Contextual Fine-Tuning

The primary advantage of contextual fine-tuning lies in its data efficiency. Unlike traditional supervised fine-tuning, which requires large, labeled datasets, contextual fine-tuning can be performed using only a handful of task-specific examples. This makes it highly attractive for domains where labeled data is sparse, expensive, or difficult to obtain.

Moreover, contextual fine-tuning allows for dynamic adaptation, where the model can be easily re-purposed for different tasks without retraining. This flexibility reduces the overhead associated with maintaining task-specific models and allows for the rapid deployment of models in new environments or with new tasks.

However, contextual fine-tuning also has its limitations. The quality of the model's output can be highly dependent on prompt design and example selection; inadequate or poorly structured prompts may lead to suboptimal performance, especially in more complex or specialized tasks [12]. Furthermore, while contextual fine-tuning is suitable for many tasks, it may not always match the performance of models that have been explicitly fine-tuned on a large, task-specific labeled dataset.

Contextual fine-tuning offers a highly flexible and efficient approach to task adaptation, allowing models to generalize across tasks with minimal labeled data. By conditioning the model with task-specific context, examples, and instructions, it is possible to guide the model's behavior and improve its performance on new and diverse tasks. In the case of text-to-SQL generation, contextual fine-tuning enables efficient adaptation to various databases and query types, making it an ideal choice for dynamic and resource-constrained enterprise systems. While it may not always

outperform supervised fine-tuning in terms of accuracy, its advantages in flexibility, efficiency, and adaptability make it a powerful tool for modern language modeling applications.

### Combining SFT with Few-Shot Prompting

A hybrid approach that combines Supervised Fine-Tuning (SFT) with few-shot prompting is gaining increasing attention in large language model adaptation [3, 11]. While SFT is an effective method for aligning the model with a specific task using labeled data, few-shot prompting enhances the mode's flexibility and adaptability, enabling it to handle a wider variety of unseen tasks without requiring retraining [10]. This combined strategy allows the model to benefit from both the precision of supervised adaptation and the flexibility of prompt-based context adjustment.

In practice, this hybrid approach operates as follows: - Fine-tuning adapts the model's parameters to the specific task, improving its overall ability to generate the desired outputs, such as SQL queries, based on a labeled training dataset. - Few-shot prompting, applied at inference time, allows the model to adjust its behavior to fit subtle task variations or domain-specific contexts by simply providing a few examples (or instructions) in the prompt.

For example, a model fine-tuned for SQL generation might be presented with a few example queries related to a specific database schema via the prompt. These examples allow the model to rapidly adapt to new queries or slight changes in the schema without requiring a full retraining. This combination of fine-tuning and contextual prompting strikes a balance between maintaining model precision (via SFT) and offering flexibility and scalability (via few-shot prompting), which is particularly valuable in real-time systems that must adapt to new tasks on-the-fly.

Moreover, combining SFT with few-shot prompting allows the model to leverage both task-specific knowledge (acquired during fine-tuning) and contextual flexibility (provided by the prompt). This enables more robust and adaptable performance in enterprise systems, where data changes frequently and new query types or schema adjustments may arise.

### Why Few-Shot Learning Remains Useful Even After Fine-Tuning

One might wonder why few-shot learning remains a valuable strategy even after the model has been fine-tuned on a specific task. After all, fine-tuning the model with a large labeled dataset typically enhances its task-specific performance, potentially reducing the need for in-context adaptation. However, there are several reasons why few-shot learning continues to be useful:

- Domain Adaptation and Schema Variability: In many enterprise systems, the underlying data schema is dynamic, and new tables, columns, or relationships

can be added over time. Fine-tuning a model on a fixed set of training data may result in overfitting to a specific schema. Few-shot learning, however, allows the model to adapt to these schema changes in real-time by simply adding new examples in the prompt. This flexibility helps mitigate the issue of schema drift, a common challenge in enterprise environments.

- Task Specialization and Flexibility: Fine-tuning a model for a particular task, such as text-to-SQL generation, results in an increase in task-specific accuracy. However, this specialized behavior may make the model less flexible when faced with tasks that deviate from its training data or involve edge cases not covered by the fine-tuning process. Few-shot learning enables the model to adapt to these novel tasks on-the-fly, using only a few prompt examples. It also provides the ability to handle a wide range of tasks that may not have been included in the fine-tuning dataset, giving the model the versatility to switch between multiple roles or domains.

- Resource Efficiency: While fine-tuning improves the model's performance for specific tasks, it is computationally expensive and requires labeled data, which may not always be available. Few-shot learning, on the other hand, can adapt the model to new tasks with minimal data and low computational overhead. This makes it particularly useful when fine-tuning is impractical or when frequent task-switching is required. In enterprise systems, where computational resources are often constrained and time-to-deployment is critical, few-shot learning offers an efficient means of task adaptation without the need for expensive retraining cycles.

- Handling Ambiguities and Variability in Input: Fine-tuning models often improve performance on well-defined, repetitive input-output mappings, such as specific types of queries or transactions. However, real-world data is often noisy and ambiguous. Few-shot learning enables the model to better handle such variability by providing contextual examples that guide the model on how to interpret or process ambiguous inputs. For instance, in the context of SQL generation, few-shot examples can clarify how to handle queries with complex joins, aggregations, or optional clauses, helping the model generate the correct SQL even in the face of novel query formulations.

- Continual Learning and Long-Term Adaptation: Few-shot learning also allows models to engage in continual learning without requiring a complete retraining. As new data or tasks emerge, few-shot prompting allows the model to quickly adapt without needing to revisit the entire training process. This is especially useful for rapidly evolving domains or systems that need to be updated frequently based on new user interactions, updated schema, or evolving business logic.

In conclusion, few-shot learning remains valuable even after fine-tuning because it enhances the model's adaptability to new or dynamic tasks, provides flexibility for handling data or schema changes, and offers a cost-effective way to achieve continual learning. When combined with SFT, few-shot prompting ensures that models can perform specialized tasks with high accuracy while retaining the ability to generalize to new, unforeseen scenarios.

## 2.3 Text-to-SQL Generation with Large Language Models

The task of translating natural language queries into structured SQL queries has been a long-standing challenge in the field of Natural Language Processing (NLP), with significant implications for improving human-computer interaction and enabling database querying through natural language interfaces. In enterprise systems, where large-scale databases store complex information, the ability to generate correct and executable SQL queries from user inputs can streamline data access and analysis.

Text-to-SQL generation involves the transformation of a natural language question or request into a corresponding SQL query that can be executed on a relational database. While early efforts in this area focused on rule-based methods or semantic parsing techniques, recent advancements have leveraged the power of large pre-trained language models to generate SQL queries in a more flexible and scalable way. These models, particularly those based on Transformer architectures, have demonstrated the ability to handle complex language constructs and generate accurate SQL queries with minimal supervision, particularly when fine-tuned on task-specific datasets.

Despite the success of large language models (LLMs) in the text-to-SQL domain, several challenges remain. These challenges primarily stem from the inherent complexity of enterprise databases, which often involve intricate schemas, relational data, and domain-specific logic. Moreover, generating SQL queries that are both syntactically correct and semantically aligned with the database schema requires sophisticated reasoning capabilities, which are often difficult for models to capture without appropriate adaptation.

This section explores the key elements of text-to-SQL generation with large language models, focusing on the problem definition, traditional approaches, and the unique challenges posed by enterprise databases. Additionally, we will discuss how LLM-based methods provide an advantage over rule-based and semantic parsing approaches, and the role of fine-tuned language models in handling complex query generation tasks.

### 2.3.1 Text-to-SQL Problem Definition

The text-to-SQL task consists of translating a natural language query expressed by a user into a structured SQL statement that can be executed against a relational database. Formally, given a natural language input $q$ and a database schema $S$, the objective is to generate an SQL query $Q_{sql}$ such that its execution over the database yields results that correctly reflect the user's intent encoded in $q$.

Unlike many natural language generation tasks, text-to-SQL requires the model to produce outputs that must satisfy strict syntactic and semantic constraints. The generated SQL query must conform to the formal grammar of the SQL language, correctly reference tables and columns defined in the schema, and encode logical relationships such as joins, filters, aggregations, and grouping operations. Any deviation from these constraints may result in a query that is either invalid or semantically incorrect.

A key aspect of the text-to-SQL problem is the need to jointly reason over two heterogeneous representations: unstructured natural language and structured relational schemas. Natural language queries often refer to database entities implicitly, using domain-specific terminology or synonyms rather than exact table or column names. Consequently, the model must perform implicit schema linking, identifying which elements of the schema correspond to the concepts expressed in the user query. This mapping is non-trivial, particularly in databases with large schemas or overlapping semantic concepts.

Another defining characteristic of the text-to-SQL task is the distinction between syntactic correctness and execution correctness. A syntactically correct SQL query adheres to the formal syntax rules of SQL and can be parsed by a database engine without errors. However, syntactic correctness alone is insufficient to guarantee that the query fulfills the user's intent. Execution correctness requires that the query, when executed on the target database, produces results that are semantically aligned with the natural language request.

For example, a query may be syntactically valid yet reference an incorrect table, apply an incorrect join condition, or omit a critical filter, leading to misleading or incomplete results. In enterprise settings, such errors can have significant consequences, as decisions may be based on incorrect data. As a result, evaluating text-to-SQL systems often emphasizes execution accuracy over surface-level string matching between predicted and reference queries [13].

The problem is further complicated by the compositional nature of SQL queries. Many real-world queries involve multiple operations combined in a hierarchical structure, such as nested subqueries, multi-table joins, and conditional aggregations. To generate such queries correctly, a model must perform multi-step reasoning, decomposing the user's request into smaller logical components and recombining

them into a coherent SQL statement. This requirement places text-to-SQL generation at the intersection of natural language understanding, symbolic reasoning, and program synthesis.

In addition, natural language queries are frequently underspecified or ambiguous. Users may omit details that are implicit in the domain context or assume shared knowledge about business logic. Resolving these ambiguities requires contextual understanding and, in some cases, assumptions about default behaviors or common analytical practices. For instance, phrases such as "top customers" or "recent performance" may require domain-specific interpretations that are not explicitly stated in the query.

From a modeling perspective, the text-to-SQL task can be viewed as a constrained sequence generation problem, where the output space is restricted by both the SQL grammar and the target schema. Unlike open-ended text generation, errors are discrete and often catastrophic: a single incorrect token may render the entire query invalid or change its semantics. This characteristic makes robustness and reliability particularly important when deploying text-to-SQL systems in production environments.

In summary, text-to-SQL generation is a complex task that requires accurate intent understanding, schema-aware reasoning, and the generation of formally correct and semantically precise SQL queries. The distinction between syntactic validity and execution correctness highlights the limitations of naïve generation approaches and motivates the need for specialized modeling techniques, fine-tuning strategies, and system-level designs that go beyond single-shot inference. These challenges are amplified in enterprise databases, where schema complexity and domain-specific semantics play a central role.

### 2.3.2 Traditional Approaches vs LLM-based Methods

The text-to-SQL task has been traditionally approached through semantic parsing and rule-based systems, long before the advent of Large Language Models. These early methods focused on explicitly modeling the mapping between natural language and formal query representations, often relying on handcrafted rules or intermediate logical forms. While such approaches laid important foundations, they exhibit limitations when applied to complex, real-world database environments.

#### Rule-Based and Template-Driven Systems

Rule-based approaches attempt to map natural language inputs to SQL queries using manually defined rules, templates, or grammars. These systems typically decompose the input query into predefined linguistic patterns and associate each pattern with a corresponding SQL construct. For example, keywords such as

"count", "average", or "top" may be mapped to aggregation functions, while simple syntactic patterns determine table selection and filtering conditions.

Although rule-based systems can perform well in constrained domains with limited query variability, they suffer from poor scalability and limited generalization. As the complexity of the database schema and the diversity of user queries increase, the number of rules required grows rapidly, making maintenance difficult and error-prone. Furthermore, these systems are highly sensitive to linguistic variability and often fail when queries deviate from expected patterns or include previously unseen formulations.

### Neural Semantic Parsing Approaches

To overcome the rigidity of rule-based systems, neural semantic parsing methods were introduced. These approaches frame text-to-SQL as a structured prediction problem, where the goal is to generate a formal query representation from natural language. Early neural models relied on sequence-to-sequence architectures with attention mechanisms, often generating an abstract syntax tree or an intermediate logical form before translating it into SQL [14].

More advanced semantic parsers incorporate explicit schema encoding, graph representations of database schemas, and constrained decoding strategies to ensure syntactic validity. While these models achieve improved generalization compared to rule-based methods, they still require carefully designed architectures and often depend on domain-specific annotations. Moreover, their performance tends to degrade as schema size and query complexity increase, particularly in enterprise-scale databases.

### Limitations of Traditional Approaches

Both rule-based and neural semantic parsing approaches share several limitations. First, they often require extensive domain-specific engineering, including hand-crafted features, grammar rules, or schema annotations. Second, they struggle to adapt to new schemas or evolving business logic without significant retraining or redesign. Finally, these approaches are typically optimized for a narrow task formulation and do not easily extend to more general reasoning or multi-step analytical workflows.

### LLM-Based Generative Methods

Large Language Models introduce a fundamentally different paradigm for text-to-SQL generation. Rather than relying on explicit rules or intermediate representations, LLM-based methods treat text-to-SQL as a generative sequence modeling task. By leveraging large-scale pre-training on diverse textual and code-like data,

LLMs acquire implicit knowledge of SQL syntax, programming constructs, and natural language semantics.

When applied to text-to-SQL, LLMs generate SQL queries directly from natural language prompts, optionally conditioned on schema information and examples. This approach offers several advantages over traditional methods. First, LLMs are inherently flexible and robust to linguistic variation, allowing them to handle a wide range of query formulations without explicit rule definitions. Second, they can adapt to new domains and schemas through prompting or fine-tuning, reducing the need for extensive manual engineering.

### Advantages of Generative LLM-Based Approaches

Generative approaches based on LLMs provide multiple benefits in practical applications:

- Schema Generalization: LLMs can generalize across different database schemas when provided with appropriate schema context, enabling reuse across multiple databases.
- Reduced Engineering Overhead: The reliance on pre-trained knowledge reduces the need for handcrafted rules or task-specific architectures.
- Compositional Reasoning: LLMs can generate complex SQL queries involving joins, aggregations, and nested subqueries through natural language reasoning.
- Integration with Natural Language Interfaces: LLMs naturally support conversational and interactive query refinement.

However, LLM-based methods are not without limitations. Generated queries may be syntactically correct yet semantically incorrect, and models may hallucinate schema elements or apply incorrect joins. These issues highlight the importance of system-level safeguards, such as schema grounding, execution feedback, and modular pipeline design.

### Toward System-Oriented Text-to-SQL Solutions

The contrast between traditional and LLM-based approaches suggests a shift from monolithic, model-centric solutions to system-oriented architectures. While LLMs offer unprecedented flexibility and expressiveness, their effective use in enterprise environments requires careful orchestration, validation, and decomposition of complex tasks. Rather than replacing traditional systems outright, LLMs are increasingly embedded within pipelines that combine classical database execution with language-based reasoning.

In summary, traditional rule-based and neural semantic parsing methods provide structured and interpretable solutions for text-to-SQL generation but struggle with scalability and adaptability. LLM-based generative approaches offer superior flexibility and generalization but introduce new challenges related to reliability and control. Addressing these challenges motivates the development of modular, orchestrated systems that leverage the strengths of LLMs while mitigating their weaknesses, as explored in the subsequent sections of this thesis.

### 2.3.3 Challenges in Enterprise Databases

Enterprise databases are typically large, complex, and highly specialized systems designed to manage vast amounts of structured data across multiple departments and use cases. These databases are central to business-critical applications, where maintaining data integrity, security, and scalability is paramount. However, the complexity of these systems presents unique challenges when applying text-to-SQL generation techniques, especially when large language models (LLMs) are employed to translate natural language queries into SQL queries that can interact with these intricate data structures.

One of the primary challenges when generating SQL queries from natural language is the complexity of the underlying database schema. In an enterprise context, databases are often composed of hundreds, if not thousands, of tables. These tables contain numerous columns, indexes, and interrelations, and the schema itself is constantly evolving to reflect the organization's shifting business needs. This dynamic nature of enterprise databases presents a problem for text-to-SQL systems, which must not only understand the structure of these schemas but also adapt to frequent changes [15]. For instance, the model must identify the relevant tables and columns from the schema when generating a SQL query, which is particularly difficult when tables store similar types of information or have overlapping column names. As the schema grows, the amount of context the model must process increases, potentially overwhelming its capacity to consider all relevant details and leading to errors or omissions in the query.

Moreover, the schema in enterprise environments is rarely static. New tables, columns, or relationships are frequently added to accommodate new business requirements. Retraining the model every time there is a change in the schema is computationally expensive and impractical. Thus, the model needs to be capable of adapting to new schema elements with minimal intervention, ideally without requiring complete retraining or fine-tuning, which adds to the operational complexity.

Another significant challenge is the need for the model to reason about joins between multiple tables. Many enterprise queries involve complex joins that link data from different tables based on shared keys or relationships. For instance,

a query might request information about customers who have made purchases in a certain period, requiring the model to join the "Customers" table with the "Orders" table. The model must identify the relevant tables and infer the correct join conditions to produce a meaningful result. The challenge becomes even more pronounced when the database schema involves many-to-many relationships or hierarchical structures. In these cases, the model must reason not only about which tables to join but also about the type of join to apply (e.g., INNER JOIN, LEFT JOIN) and the appropriate conditions for joining the tables. A failure to generate the correct join or to identify the proper relationship can result in incorrect or incomplete results, or the query may fail to execute entirely.

Enterprise databases are also characterized by their reliance on domain-specific business logic and semantics. Business logic refers to the rules and processes that govern how data is collected, processed, and used within an organization. These rules are often implicit in the schema or the application's logic but must be taken into account when generating queries. For example, a query might need to consider specific business constraints, such as calculating discounts for products based on customer loyalty or ensuring that only data from active customers is retrieved. Domain semantics, on the other hand, involves the meaning of terms and concepts within the context of a specific industry. A term like "revenue" in an e-commerce system may have a different meaning from "revenue" in a financial database. Thus, understanding these domain-specific terms and applying the correct logic in the SQL query is essential for the model to generate accurate and meaningful results. The model must be able to recognize these terms and map them to the appropriate tables and columns in the schema, often requiring knowledge that is outside the scope of general language modeling.

In addition to schema complexity and business logic, handling complex queries and large result sets also presents challenges. Queries in enterprise databases often involve aggregations, filtering conditions, and multi-table joins, which can lead to large result sets or lengthy query execution times. This is particularly problematic when the database contains millions of rows of data. Generating SQL queries that efficiently retrieve only the relevant data, without over-fetching or causing delays, is a crucial aspect of the task. Moreover, SQL queries must be optimized to ensure they execute quickly and with minimal resource usage, particularly in enterprise environments where the system needs to handle high volumes of concurrent queries.

Another major challenge is ensuring that the system is capable of scaling to meet the demands of large enterprise systems. These systems must process a large volume of queries in real-time, often with stringent latency requirements. The text-to-SQL model must not only generate accurate queries but also execute them in a timely manner, even when the query load is high. This requires careful optimization of both the model's inference time and the database's execution time. As such, the ability to generate fast and efficient queries becomes critical in real-time systems

that must respond quickly to user requests, particularly when dealing with vast and complex datasets.

Finally, real-time requirements in enterprise environments introduce another layer of complexity. The integration of text-to-SQL models into live, operational systems demands that these models be both accurate and responsive. Enterprises rely on fast decision-making and automated systems that can generate complex queries on-the-fly, without delays. As a result, the text-to-SQL system must be capable of handling a variety of dynamic, real-time queries, adapting to changes in user requests, and scaling across different database environments with minimal downtime.

In summary, the challenges of text-to-SQL generation in enterprise environments stem from the complex nature of the database schema, the intricate relationships between tables, and the need to incorporate domain-specific business logic into query generation. These challenges are further compounded by the need for real-time performance, scalability, and the ability to handle large and dynamic datasets. Overcoming these challenges requires robust, flexible models that can reason about complex relationships and generate accurate queries, all while efficiently scaling to meet the needs of real-time enterprise systems. Addressing these issues is crucial for deploying effective text-to-SQL systems in enterprise environments, and forms the basis for the architectural strategies and system designs explored in subsequent chapters of this thesis.

## 2.4 Decomposition and Modular Reasoning in LLM Systems

The limitations of monolithic LLM inference motivate the adoption of decomposed and modular reasoning strategies, in which complex tasks are broken down into smaller, more manageable subtasks. This paradigm aligns with long-standing principles in artificial intelligence and cognitive science, where problem decomposition and hierarchical planning are used to reduce complexity, improve interpretability, and enhance robustness.

In the context of Large Language Models, decomposition enables the separation of high-level reasoning from low-level execution, allowing each stage of the task to be handled more effectively and with greater reliability.

### 2.4.1 Problem Decomposition in Artificial Intelligence

Problem decomposition refers to the process of partitioning a complex task into a set of simpler subtasks whose solutions can be combined to form a complete solution. Formally, a task $T$ can be expressed as:

$$T = \{t_1, t_2, \ldots, t_n\}$$

where each subtask $t_i$ addresses a specific aspect of the overall problem.

This divide-and-conquer strategy has been widely used in classical AI systems, including:

- hierarchical planning algorithms,
- expert systems,
- modular software architectures,
- symbolic reasoning pipelines.

Decomposition reduces cognitive and computational complexity by constraining the scope of each reasoning step. Rather than solving a high-dimensional problem in a single inference pass, the system solves multiple lower-dimensional problems, each with clearer objectives and tighter constraints.

In analytical tasks such as database querying, decomposition enables the separation of:

- data retrieval,
- aggregation,
- anomaly detection,
- trend analysis,
- result synthesis.

Each component can be optimized independently, improving overall system reliability.

### 2.4.2 Multi-Step Reasoning with Large Language Models

LLMs are inherently sequential, generating outputs token by token based on previous context. While this enables fluent text generation, it does not guarantee structured reasoning over multiple conceptual steps.

Multi-step reasoning introduces an explicit intermediate structure to the inference process. Instead of producing a final answer in a single pass, the model is guided to generate intermediate reasoning states:

$$x \to r_1 \to r_2 \to \cdots \to y$$

where $x$ is the input, $r_i$ are intermediate reasoning steps, and $y$ is the final output.

Techniques such as chain-of-thought prompting encourage models to externalize intermediate reasoning steps, often improving performance on tasks that require logical decomposition, arithmetic, or symbolic manipulation [16].

However, purely textual multi-step reasoning remains fragile, as intermediate steps are not validated against external sources or execution environments. This limitation motivates tool-augmented and agentic approaches, where the model can interleave reasoning with actions such as querying databases or calling external utilities [17, 18].

### 2.4.3 Separation of Planning and Execution

In complex systems, it is often beneficial to separate planning from execution. Planning involves deciding what actions to take, while execution involves carrying out those actions and observing their outcomes.

In LLM-based pipelines, this corresponds to:

- Planning: task decomposition, query formulation, strategy selection;
- Execution: SQL generation, query execution, result retrieval.

This separation reduces the cognitive load placed on a single model invocation and allows different models to specialize in different roles. Planning models can focus on high-level reasoning and abstraction, while execution models can be optimized for precision and syntactic correctness.

Furthermore, execution results can be fed back into the planning stage, enabling iterative refinement and error correction.

#### Modular Reasoning Pipelines

Modular pipelines organize the reasoning process into discrete components connected by structured interfaces. Each module performs a specific function, such as:

- query generation,
- execution,
- result validation,
- follow-up planning,
- final synthesis.

Formally, a modular pipeline can be represented as a directed graph:

$$M_1 \rightarrow M_2 \rightarrow \cdots \rightarrow M_k$$

where each module $M_i$ processes the output of the previous module.
This structure offers several advantages:

- improved interpretability,
- easier debugging,
- localized error handling,
- flexible system reconfiguration.

In Text-to-SQL systems, modularity allows SQL generation to be isolated from analytical reasoning, reducing the risk that errors in one component propagate uncontrollably through the system.

### 2.4.4 Parallelism and Task Specialization

Decomposition also enables parallel execution of independent subtasks. For example, different analytical queries can be executed simultaneously to retrieve complementary views of the data.

Parallelism improves system throughput and reduces end-to-end latency, particularly when individual subtasks involve external calls such as database queries.

Moreover, task specialization allows each module to be optimized for its specific function. For instance:

- a fine-tuned model can be used for SQL generation,
- a general-purpose LLM can be used for reasoning and synthesis.

This role-based specialization improves both accuracy and efficiency.

#### Towards Agent-Based Architectures

When modular pipelines incorporate autonomous decision-making, feedback loops, and conditional execution, they naturally evolve into agent-based architectures.

In such systems, an orchestrator model coordinates multiple worker models, deciding:

- which subtasks to execute,
- when to request additional information,

- how to integrate intermediate results.

This agent-based paradigm enables:

- iterative exploration of data,
- dynamic refinement of hypotheses,
- automatic error recovery,
- adaptive analytical strategies.

These properties make agent-based systems particularly suitable for complex analytical tasks, where insights emerge through successive interactions with structured data sources.

The following section introduces orchestration frameworks that support such architectures, providing the technical foundation for implementing multi-agent reasoning pipelines in practice.

### 2.4.5 Agent-Based and Orchestrated LLM Architectures

Agent-based and orchestrated architectures treat LLM problem solving as a sequence of decisions, actions, and observations, where the model interacts with an external environment (e.g., databases, APIs, execution engines) and updates its state based on tool feedback. This design improves robustness by grounding intermediate steps in executable evidence and is commonly implemented by interleaving natural language reasoning with explicit tool calls [17, 18].

### 2.4.6 The Orchestrator–Worker Paradigm

#### Role Separation

A central design principle in modern LLM systems is the separation of roles between high-level control and low-level execution. Rather than relying on a single model to perform all reasoning and operational tasks, the system is decomposed into an orchestrator and one or more workers. The orchestrator is responsible for interpreting the user's intent, decomposing complex requests into manageable subtasks, deciding which operations to perform, and synthesizing the final response. In contrast, workers are specialized components that execute well-defined subtasks such as generating SQL queries, running code, or retrieving data from external sources.

This separation of responsibilities improves robustness by isolating execution errors from the reasoning process. If a worker fails to complete a task correctly, the orchestrator can detect the failure and issue corrective instructions without

compromising the entire reasoning pipeline. Moreover, role separation enhances interpretability by making intermediate steps explicit, allowing the system's decision-making process to be inspected and analyzed.

#### Control vs Execution Models

The distinction between control and execution roles also reflects different computational requirements. Orchestrator models must excel at instruction following, abstract reasoning, task decomposition, and long-context synthesis. They operate primarily in the space of high-level planning and interpretation. Worker models, on the other hand, are optimized for precision, consistency, and efficiency in narrow domains. Their primary function is to execute specific operations reliably rather than to perform open-ended reasoning.

This division enables the system to assign computationally expensive reasoning tasks to powerful models while delegating repetitive or structured operations to more efficient components. As a result, overall system performance improves not only in terms of accuracy, but also in latency, scalability, and cost-efficiency.

### 2.4.7 LLMs as Controllers and Executors

#### Reasoning Models vs Task-Specific Models

In orchestrated LLM systems, different models can be specialized for distinct cognitive roles. Reasoning-oriented models are designed to handle abstract problem formulation, multi-step planning, and synthesis of heterogeneous information. These models are typically trained or aligned to follow complex instructions, reason over partial evidence, and generate coherent analytical narratives.

Task-specific models, in contrast, focus on narrow execution-oriented objectives. Examples include models fine-tuned for SQL generation, code completion, or structured data extraction. Their outputs are evaluated primarily on correctness, syntactic validity, and domain compliance rather than on explanatory depth. By constraining the task scope, these models achieve higher reliability in their respective domains.

The interaction between reasoning and task-specific models mirrors a cognitive pipeline in which high-level intentions are translated into concrete operations. The orchestrator formulates an abstract plan, while the workers implement the plan through precise, domain-specific actions.

#### Why Different Models for Different Roles

Using different models for different roles addresses fundamental trade-offs in LLM design. Large, general-purpose models offer strong reasoning capabilities but incur

high computational costs and latency. Smaller or fine-tuned models are more efficient but lack broad reasoning flexibility. By combining them within a single system, it is possible to exploit their complementary strengths.

This modularity also improves system robustness. Errors in execution can be localized to worker components, while reasoning errors can be mitigated through iterative planning and validation. Furthermore, role-based specialization enables easier system maintenance and upgrading: individual components can be replaced or improved without redesigning the entire architecture.

### 2.4.8 Graph-Based Orchestration Frameworks

#### DAG-Based Execution

As reasoning pipelines grow in complexity, linear sequences of model calls become insufficient. Graph-based orchestration frameworks address this limitation by representing reasoning processes as directed acyclic graphs (DAGs). In this representation, nodes correspond to reasoning states or actions, while edges encode the flow of control and information between them.

DAG-based execution allows the system to branch into multiple reasoning paths, merge partial results, and manage dependencies between subtasks. This structure is particularly well suited for analytical workflows, where different aspects of a problem can be explored independently before being integrated into a unified interpretation.

#### Conditional Flows and Retries

Graph-based orchestration also enables conditional execution. Instead of following a fixed sequence of steps, the system can adapt its behavior based on intermediate outcomes. If a subtask fails or produces incomplete results, the orchestrator can trigger alternative actions such as reformulating a query, requesting additional data, or revising earlier assumptions.

Retry mechanisms further enhance reliability by allowing the system to correct errors automatically. Rather than terminating on failure, the system treats errors as informative signals that guide subsequent reasoning. This iterative refinement process resembles a feedback loop in which the model continuously adjusts its actions based on observed outcomes.

#### Parallelism and Fault Tolerance

Another advantage of graph-based orchestration is the ability to execute independent subtasks in parallel. By distributing multiple worker calls concurrently, the system reduces overall latency and improves throughput. Parallel execution is particularly

beneficial in analytical scenarios that require multiple independent data queries or computations.

Fault tolerance is also improved through modular execution. If one branch of the reasoning graph fails, other branches can still complete successfully, allowing the orchestrator to synthesize partial results. This resilience is crucial for real-world deployments, where external tools and data sources may be unreliable.

Graph-based orchestration frameworks therefore provide the structural foundation for scalable, adaptive, and robust LLM-driven systems. They enable complex reasoning workflows that extend beyond the capabilities of single-pass inference, paving the way for practical applications in data analysis, decision support, and automated knowledge discovery.

### Iterative Reasoning and Self-Investigation Loops

Advanced agent-based systems often implement iterative loops in which the orchestrator evaluates intermediate results, decides whether additional evidence is needed, and triggers follow-up actions (e.g., reformulating a query, adding filters, or requesting diagnostic sub-queries). This closed-loop structure improves robustness because errors and ambiguities can be detected and corrected using execution feedback rather than relying on a single generative pass.

In practice, iterative loops improve interpretability by producing a trace of intermediate artifacts (hypotheses, tool calls, results, and revisions) and they provide a natural mechanism for uncertainty management: the system can continue gathering evidence until stopping criteria are met (e.g., stable results or budget limits). [17]

# Chapter 3

# Experimental Setup and Evaluation

## 3.1 Goals and Research Questions

The primary objective of this experimental study is to systematically investigate the effectiveness of different adaptation strategies for Large Language Models applied to the Text-to-SQL task in a realistic enterprise-like database setting. In particular, the experiments aim to compare prompt-based and training-based approaches for enabling accurate SQL generation over a fixed relational schema, with an explicit focus on robustness, scalability, and deployment feasibility.

The study is conducted on the Northwind database, chosen as a representative example of a structured business-oriented schema involving multiple interrelated entities, foreign key constraints, and domain-specific logic. The experiments focus on complex analytical queries expressed in natural language, requiring non-trivial joins, aggregations, temporal reasoning, and compliance with strict SQL dialect constraints (MySQL 8). This setting allows the evaluation of model behavior beyond simple lookup queries, emphasizing realistic failure modes encountered in production systems.

### 3.1.1 Experimental Scope

The experimental scope is centered on the comparison of three broad classes of adaptation strategies:

1. Contextual Fine-Tuning (Contextual FT), where the base model is not trained on domain-specific data, but is instead provided at inference time with detailed contextual information. This context includes the full database schema, explicit

foreign key relationships, domain-specific business rules, and SQL dialect constraints encoded directly in the prompt.

2. Supervised Fine-Tuning (SFT), where the model is explicitly trained on a dataset of natural language questions paired with correct SQL queries specific to the target database. In this setting, the model is expected to internalize both the schema structure and query patterns during training, thereby reducing reliance on large prompts at inference time.

3. Hybrid approaches combining SFT with prompt-based techniques, including rule-based prompting and few-shot learning. These methods aim to exploit the strengths of both paradigms by using supervised training for schema specialization while retaining prompt-level control for enforcing syntactic constraints and demonstrating complex reasoning patterns.

The comparison is carried out using the same base model family (GPT-4.1 variants) to isolate the effects of adaptation strategy from differences in model architecture or pretraining. Special attention is given to the GPT-4.1 Mini variant, which represents a realistic choice for deployment scenarios where cost and latency constraints are critical.

### 3.1.2 Motivation and Research Challenges

While contextual prompting has demonstrated strong performance in prior work, it introduces significant overhead at inference time. Providing full database schemas, constraint descriptions, and domain rules in every request increases token consumption, latency, and operational cost. Moreover, long prompts approach the limits of the context window and increase the risk of prompt truncation or degraded performance.

Conversely, supervised fine-tuning promises more compact inference-time interactions by embedding domain knowledge directly into the model's parameters. However, SFT introduces its own challenges, including the need for curated training datasets, sensitivity to dataset size and composition, and the risk of overfitting or incomplete generalization. In the Text-to-SQL domain, an additional difficulty arises from the need to enforce strict SQL dialect rules that may differ from the generic SQL patterns learned during pretraining.

The experiments are designed to explore these trade-offs explicitly, assessing whether supervised fine-tuning alone is sufficient to replace contextual prompting, and under which conditions hybrid strategies provide measurable benefits.

### 3.1.3 Research Questions

Based on these considerations, the experimental evaluation is guided by the following research questions:

- RQ1: How does contextual fine-tuning compare to supervised fine-tuning in terms of SQL correctness on complex analytical queries?
- RQ2: To what extent can supervised fine-tuning internalize database-specific schema knowledge and reduce reliance on explicit schema descriptions at inference time?
- RQ3: Are the failure modes observed in supervised fine-tuning primarily related to query complexity or to violations of SQL dialect-specific constraints?
- RQ4: Does combining supervised fine-tuning with inference-time prompting (rules and few-shot examples) lead to superior performance compared to either approach in isolation?
- RQ5: How do different adaptation strategies trade off accuracy, inference cost, and latency, and which configurations are most suitable for scalable deployment?

These questions aim to disentangle the relative contributions of training data, prompt engineering, and model capacity to overall system performance.

### 3.1.4 Hypotheses

The experiments are structured to test the following hypotheses:

- H1: Contextual fine-tuning provides strong baseline performance on complex Text-to-SQL tasks due to the explicit availability of schema and rule information, but incurs high inference-time costs.
- H2: Supervised fine-tuning on a limited dataset enables the model to internalize database structure but is insufficient to reliably enforce complex SQL dialect constraints without additional guidance.
- H3: Increasing the number of supervised fine-tuning samples improves performance, but exhibits diminishing returns when attempting to override pre-trained syntactic biases.
- H4: Hybrid approaches that combine supervised fine-tuning with inference-time prompting and few-shot examples outperform both pure contextual fine-tuning and pure supervised fine-tuning in terms of accuracy on complex queries.

- H5: The combination of supervised fine-tuning and few-shot prompting achieves comparable or superior accuracy to contextual fine-tuning while significantly reducing inference-time token usage and operational cost.

Together, these hypotheses reflect the central claim of this work: that optimal performance in practical Text-to-SQL systems emerges not from a single adaptation technique, but from a carefully designed combination of training-based specialization and prompt-based control mechanisms.

### 3.1.5 Expected Contributions

By addressing these research questions, this chapter aims to provide empirical evidence supporting design principles for LLM-based Text-to-SQL systems deployed in enterprise environments. In particular, the experiments seek to clarify when supervised fine-tuning is beneficial, when prompt-based adaptation remains necessary, and how the two can be effectively integrated to achieve both high accuracy and operational efficiency.

The findings of this evaluation serve as the empirical foundation for the subsequent introduction of a multi-agent orchestration pipeline, where specialized Text-to-SQL components are embedded within a larger analytical reasoning system. After selecting the best-performing configuration from the comparative study, the chosen setup was integrated into an orchestrated pipeline in which query understanding, SQL generation, execution, and result validation are handled as distinct stages coordinated by an orchestrator. This design ensures that the evaluation outcomes translate into an end-to-end system that is robust to execution errors and suitable for realistic deployment.

## 3.2 Dataset and Test Environment

This section describes the dataset, database schema, and experimental environment used throughout the evaluation. Particular attention is given to the characteristics of the selected database, the rationale behind its choice, and the technical setup employed for training and inference. These details are essential to ensure reproducibility and to contextualize the observed experimental results.

### 3.2.1 Northwind Database Overview

The experiments are conducted using the Northwind database, a well-established relational dataset originally designed to model a small-to-medium-sized enterprise engaged in order management and product distribution. Northwind is widely adopted as a benchmark for database-related tasks due to its realistic structure,

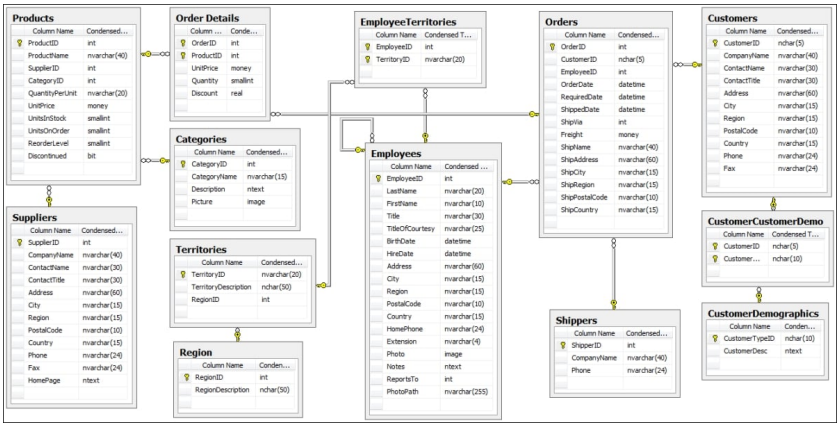


**Figure 3.1:** Entity–relationship (ER) model of the Northwind database schema.

normalized schema, and clear representation of business processes such as order fulfillment, inventory management, and customer relationships.

Figure 3.1 presents the entity–relationship (ER) model of the Northwind schema used throughout the evaluation.

The database schema consists of multiple interrelated tables, each representing a distinct business entity. Core tables include `Customers`, `Orders`, `OrderDetails`, `Products`, `Categories`, `Suppliers`, `Employees`, and `Shippers`. Together, these tables capture the end-to-end lifecycle of sales transactions, from customer acquisition to product shipment.

Primary keys are defined on all entity tables to ensure unique identification of records. For instance, `Customers(CustomerID)`, `Orders(OrderID)`, and `Products(ProductID)` serve as unique identifiers for their respective entities. Referential integrity is enforced through a set of foreign key constraints that model real-world relationships. Notable examples include the linkage between `Orders` and `Customers` via `CustomerID`, between `OrderDetails` and `Orders` via `OrderID`, and between `Products` and `Categories` via `CategoryID`.

This relational structure introduces non-trivial join paths that are particularly relevant for Text-to-SQL evaluation. Many analytical queries require reasoning across multiple tables, combining transactional data with categorical and temporal dimensions. The presence of junction tables such as `OrderDetails`, which encode many-to-many relationships between orders and products, further increases query complexity and makes Northwind an appropriate testbed for evaluating join reasoning and aggregation correctness.

### 3.2.2 Business Logic and Analytical Complexity

Beyond its relational structure, Northwind encodes implicit business logic that must be respected during query generation. Revenue calculations, for example, require combining quantity, unit price, and discount information stored at the order-line level. Temporal analysis often involves grouping orders by month, quarter, or year, while customer and product performance analyses require aggregation across multiple dimensions.

These characteristics make the database particularly suitable for evaluating complex analytical Text-to-SQL queries, as they expose common sources of error such as incorrect aggregation levels, invalid grouping under strict SQL modes, or misuse of joins that lead to duplicated rows. Consequently, Northwind serves not only as a structural benchmark, but also as a semantic and logical stress test for SQL generation systems.

### 3.2.3 Synthetic Data Generation

In this work, the experiments focus primarily on the structural and logical correctness of generated SQL queries rather than on the statistical properties of the underlying data. As such, the default Northwind dataset is sufficient to exercise the relevant query patterns. Where necessary, the database was populated or extended with synthetic data to ensure coverage of edge cases, such as sparse categories, low-frequency customers, or temporally skewed order distributions. Synthetic records were generated using the `Faker` library, a widely used data-generation toolkit that produces realistic-looking but fictitious values (e.g., names, addresses, dates, and company information) according to locale-specific templates. This approach enables the creation of plausible data for testing and evaluation while avoiding the use of sensitive or personally identifiable information.

Synthetic data generation followed schema-preserving procedures to maintain referential integrity. New records were generated in a way that respected primary and foreign key constraints, ensuring that all joins remained valid and that no orphaned records were introduced. The use of synthetic data allowed the creation of controlled scenarios for testing edge cases without altering the logical structure of the database.

### 3.2.4 Natural Language Query Language

All natural language queries used in both training and evaluation are expressed in Italian. This choice introduces an additional layer of complexity compared to English-only benchmarks, as it requires the model to correctly map non-English linguistic structures to database concepts and SQL syntax.

The use of Italian queries also tests the multilingual generalization capabilities of the underlying language models and ensures that the evaluation reflects realistic usage scenarios in non-English enterprise environments. Importantly, while the natural language inputs are in Italian, all generated SQL queries adhere strictly to the MySQL 8 dialect and reference English table and column names as defined in the database schema.

### 3.2.5 Database Engine and SQL Dialect

All experiments are executed on MySQL 8, configured with strict SQL modes enabled, including `ONLY_FULL_GROUP_BY`. This configuration enforces rigorous compliance with SQL standards and exposes a class of errors that are often overlooked in more permissive database settings.

The strict SQL configuration is intentionally chosen to stress-test the generated queries and to evaluate the model's ability to adhere to dialect-specific constraints. Common failure modes in this setting include improper grouping, invalid use of aggregate functions, and ambiguous column references. These constraints play a central role in the experimental evaluation and directly motivate the use of both training-based and prompt-based guardrails.

### 3.2.6 Training and Inference Environment

Supervised fine-tuning experiments are conducted using OpenAI's fine-tuning infrastructure, following the JSONL-based format required for model adaptation. Training datasets consist of natural language prompts and corresponding SQL queries, with system messages providing high-level task context. Hyperparameters such as learning rate, number of epochs, and batch size are selected according to recommended defaults to minimize overfitting given the relatively small dataset sizes.

Inference is performed using the OpenAI API, with different model variants (GPT-4.1 and GPT-4.1 Mini) evaluated under identical conditions to ensure comparability. API access is authenticated via an API key, which enables programmatic submission of requests from the experimental pipeline. Consistent with the OpenAI API interaction model, each request is treated as stateless: conversational context is not retained by default across calls, and any required history or task context must be explicitly included in the prompt. Prompt construction therefore varies across experimental configurations, ranging from minimal system messages in fine-tuned models to extensive contextual prompts in the contextual fine-tuning baseline.

The experimental pipeline is implemented in Python, leveraging standard libraries for database interaction and concurrency management. SQL queries are executed via a MySQL connector, and parallel execution of independent queries is

achieved using a thread-based executor. This setup enables realistic measurement of latency and throughput under concurrent workloads.

### 3.2.7 Reproducibility and Versioning

All experiments are conducted using fixed random seeds where applicable, and model versions are explicitly specified to ensure reproducibility. Database schema definitions, training datasets, and evaluation queries are version-controlled, allowing the experimental setup to be reconstructed and extended in future work.

By clearly specifying the dataset characteristics and experimental environment, this section establishes a solid foundation for interpreting the experimental results presented in subsequent sections.

## 3.3 Complexity Criteria and Query Design

A central aspect of the experimental evaluation is the definition and control of query complexity. Since the Text-to-SQL task admits a wide range of difficulty levels, from simple single-table lookups to multi-stage analytical queries, it is essential to formalize what constitutes a complex query in the context of this study. This section introduces a set of complexity criteria designed to characterize the structural, semantic, and syntactic challenges posed by the evaluation queries.

The adopted criteria aim to go beyond superficial measures such as query length or number of tokens. Instead, complexity is defined in terms of the reasoning capabilities required to generate a correct and executable SQL query under strict MySQL 8 constraints. These criteria are used both to design the test set and to interpret model failure modes observed during evaluation.

### 3.3.1 Dimensions of Query Complexity

Query complexity in this work is modeled as a multi-dimensional construct. Each query is characterized along several orthogonal dimensions, reflecting different sources of difficulty for Large Language Models. These dimensions are not mutually exclusive and often co-occur within the same query.

#### Structural Complexity

Structural complexity refers to the number and type of relational operations required to produce the correct result. Queries with high structural complexity typically involve multiple tables connected through foreign key relationships, requiring the model to correctly infer join paths and join conditions.

In the Northwind schema, structural complexity arises naturally from the presence of junction tables such as `OrderDetails`, which mediate many-to-many relationships between orders and products. Queries that require aggregating product-level information at the order, customer, or category level necessitate multiple joins with precise join predicates. Errors in this dimension often manifest as missing joins, incorrect join conditions, or unintended row duplication due to improper join ordering.

Structural complexity is particularly challenging for LLMs because it requires maintaining a global view of the schema while simultaneously reasoning about local join semantics.

### Aggregation and Grouping Complexity

Aggregation complexity captures the difficulty associated with correctly applying aggregate functions and grouping clauses. Analytical queries frequently require computing sums, averages, counts, or derived metrics such as revenue. In MySQL 8, the presence of the `ONLY_FULL_GROUP_BY` mode enforces strict compliance: every non-aggregated column in the `SELECT` clause must appear in the `GROUP BY` clause.

Queries that aggregate over multiple dimensions—such as revenue by category and year—require careful alignment between selected columns and grouping expressions. LLMs often fail in this dimension by selecting extraneous columns, grouping at the wrong level of granularity, or applying aggregates after joins that inflate row counts.

This dimension is critical for evaluating the model's ability to reason about aggregation semantics rather than merely producing syntactically valid SQL.

### Temporal Reasoning Complexity

Temporal complexity arises when queries involve time-based filtering, grouping, or trend analysis. Examples include computing year-over-year growth, identifying seasonal patterns, or aggregating metrics by month or quarter.

Such queries require the model to correctly manipulate date and time fields, extract temporal components, and apply appropriate grouping logic. Errors in temporal reasoning often involve incorrect date functions, misaligned time windows, or improper handling of boundary conditions (e.g., inclusive vs exclusive ranges).

Temporal complexity is particularly relevant for enterprise analytics and serves as a strong indicator of a model's ability to perform higher-level analytical reasoning.

### Business Logic and Derived Metrics

Many evaluation queries require the computation of derived metrics that are not explicitly stored in the database. A canonical example in Northwind is revenue

calculation, which depends on quantity, unit price, and discount values stored at the order-line level.

Correctly computing such metrics requires the model to apply domain-specific formulas consistently and at the appropriate stage of the query. Errors in this dimension include applying the formula at the wrong aggregation level, omitting discount factors, or combining incompatible units.

This dimension evaluates the model's capacity to integrate implicit business logic into SQL generation, a requirement that is rarely addressed by purely syntactic benchmarks.

#### Conditional Logic and Filtering

Conditional complexity captures the use of non-trivial filtering conditions, including nested predicates, threshold-based selections, and comparisons against aggregated values. Queries that identify top-performing entities, outliers, or anomalies often rely on `HAVING` clauses, subqueries, or conditional expressions.

LLMs frequently struggle to distinguish between row-level filters (`WHERE`) and group-level filters (`HAVING`), leading to logically incorrect queries that nevertheless execute without error. This dimension is therefore essential for evaluating semantic correctness beyond execution success.

#### Dialect-Specific Constraints and Syntax

Finally, dialect-specific complexity accounts for constraints imposed by the MySQL 8 SQL dialect. These include restrictions on the interaction between window functions and grouping, requirements for aliasing subqueries and common table expressions, and limitations on column visibility across query scopes.

Many of these constraints are subtle and are not explicitly encoded in the SQL grammar alone. Instead, they reflect execution-time semantics that must be respected to avoid runtime errors. This dimension is particularly relevant for assessing the effectiveness of prompt-based guardrails and supervised fine-tuning in enforcing dialect compliance.

### 3.3.2 Query Design Methodology

The evaluation queries are designed to systematically exercise the complexity dimensions described above. Rather than sampling queries randomly, the test set is curated to include a diverse range of analytical tasks that reflect realistic enterprise use cases.

Each query is constructed to involve multiple complexity dimensions simultaneously. For example, a query may require multi-table joins (structural complexity), revenue aggregation (business logic), temporal grouping (temporal complexity), and

strict compliance with `ONLY_FULL_GROUP_BY` (dialect-specific complexity). This design choice ensures that high performance cannot be achieved through partial reasoning or template-based generation alone.

### 3.3.3 Complexity Annotation and Coverage

To facilitate systematic analysis, each query in the test set is annotated with the complexity dimensions it involves. This annotation enables post-hoc analysis of model performance as a function of query characteristics, allowing failure modes to be correlated with specific sources of complexity.

The final test set consists of an initial collection of 30 complex queries, later extended with an additional 10 queries to further stress-test the most challenging configurations. This extension is used to validate the stability of the observed performance trends and to assess generalization beyond the original test set.

### 3.3.4 Role of Complexity in Experimental Interpretation

The explicit modeling of query complexity plays a critical role in interpreting the experimental results. Rather than treating all errors as equivalent, the analysis distinguishes between failures arising from schema misunderstanding, aggregation mistakes, business logic violations, and dialect-specific syntax errors.

This distinction is particularly important when comparing contextual fine-tuning and supervised fine-tuning approaches. While both methods may succeed on simpler queries, their behavior diverges significantly as complexity increases. By grounding the evaluation in a structured complexity framework, the experiments provide deeper insight into the strengths and limitations of each adaptation strategy.

In summary, the complexity criteria defined in this section establish a rigorous foundation for evaluating Text-to-SQL performance in realistic analytical settings. They enable a nuanced interpretation of experimental outcomes and directly inform the design of hybrid adaptation strategies and orchestrated pipelines explored in subsequent sections.

### 3.3.5 Formalization of Query Complexity Dimensions

To ensure a systematic and reproducible characterization of query difficulty, the qualitative dimensions of complexity introduced in the previous section are formalized into a set of discrete, interpretable criteria. Each evaluation query is annotated along six independent complexity dimensions, reflecting distinct sources of difficulty for Text-to-SQL generation in enterprise databases.

The purpose of this formalization is twofold. First, it enables controlled construction of the evaluation set by ensuring balanced coverage of different complexity

sources. Second, it supports post-hoc analysis by allowing model failures to be correlated with specific types of reasoning challenges rather than treated as undifferentiated errors.

### Structural Complexity

Structural complexity captures the difficulty associated with relational reasoning and join construction. It is defined based on the number of tables involved and the nature of their relationships.

Each query is assigned a structural complexity level according to the following scale:

- Level 0: Single-table query with no joins.
- Level 1: Two-table join via a direct foreign key.
- Level 2: Multi-table joins involving at least one junction table.
- Level 3: Multi-hop joins across three or more entity types, possibly involving both transactional and dimensional tables.

This dimension isolates the model's ability to reason over schema topology and correctly infer join paths.

### Window and Advanced Aggregation Complexity

This dimension captures the use of advanced aggregation mechanisms, including window functions and nested aggregations.

Queries are classified as:

- Level 0: No aggregation or simple aggregate without grouping.
- Level 1: Standard aggregation with `GROUP BY`.
- Level 2: Aggregation combined with `HAVING` or subqueries.
- Level 3: Use of window functions or combinations of window functions and grouping.

This criterion reflects the difficulty of correctly sequencing aggregation operations and respecting SQL execution semantics.

### Temporal Reasoning Complexity

Temporal complexity measures the extent to which a query requires reasoning over time dimensions.

Levels are assigned as follows:

- Level 0: No temporal component.
- Level 1: Simple filtering on date ranges.
- Level 2: Temporal grouping (e.g., by month, quarter, or year).
- Level 3: Comparative temporal analysis (e.g., year-over-year growth, trend detection).

This dimension evaluates the model's ability to manipulate date attributes and align temporal granularity with aggregation logic.

### Business Logic and Derived Metric Complexity

Business logic complexity captures the need to compute derived metrics that are not explicitly stored in the database.

Queries are categorized as:

- Level 0: Direct retrieval of stored values.
- Level 1: Simple arithmetic combinations of columns.
- Level 2: Domain-specific formulas applied at a single aggregation level.
- Level 3: Derived metrics combined with multi-level aggregation or conditional logic.

In the Northwind database, this dimension is primarily exercised through revenue computations that involve quantity, unit price, and discount factors.

### Conditional Classification and Filtering Complexity

This dimension captures the complexity of conditional logic required to classify, filter, or rank entities.

Levels are defined as:

- Level 0: Simple row-level filtering using `WHERE`.
- Level 1: Group-level filtering using `HAVING`.
- Level 2: Conditional classification using thresholds or categorical rules.

- Level 3: Nested conditions, ranking, or outlier detection.

This criterion evaluates the model's ability to distinguish between row-level and aggregate-level semantics.

#### MySQL Dialect and Constraint Complexity

This dimension captures difficulty arising from MySQL 8-specific syntax and execution constraints.

Queries are classified as:

- Level 0: No dialect-specific constraints.
- Level 1: Strict `ONLY_FULL_GROUP_BY` compliance.
- Level 2: Subqueries or common table expressions with scope constraints.
- Level 3: Interactions between grouping, window functions, and alias visibility.

This dimension is particularly important for distinguishing syntactic failures from semantic reasoning errors.

### 3.3.6 Complexity Annotation and Usage

Each query in the evaluation set is annotated with a six-dimensional complexity vector

$$\mathbf{c} = (c_s, c_w, c_t, c_b, c_f, c_m)$$

where each component corresponds to one of the complexity dimensions described above.

These annotations are not directly used to compute a single scalar difficulty score. Instead, they serve as descriptive metadata that enables stratified analysis of model performance. This design choice avoids collapsing heterogeneous sources of difficulty into a single number, preserving interpretability while still allowing fine-grained evaluation.

The annotated complexity vectors are later used in error analysis to identify systematic weaknesses of different adaptation strategies and to explain performance differences observed across experimental configurations.

## 3.4 Evaluation Protocol and Metrics

This section describes the evaluation protocol adopted to assess the performance of the different model configurations considered in this work, together with the metrics used to quantify correctness, robustness, and efficiency. The protocol is designed

to ensure a fair and reproducible comparison between contextual fine-tuning, supervised fine-tuning, and hybrid approaches combining SFT with prompt-based techniques such as few-shot learning.

### 3.4.1 Test Set Definition and Execution Repetition

Model evaluation is conducted on a curated test set composed of 40 complex natural language queries expressed in Italian. The core evaluation set consists of 30 queries, which are used consistently across all experiments. These queries are designed to stress multiple dimensions of complexity, including multi-table joins, advanced aggregations, temporal reasoning, and business-specific logic.

An additional set of 10 queries is introduced exclusively in the final comparison phase between the most effective configurations, namely the SFT+prompt+few-shot approach and the contextual+few-shot baseline. These supplementary queries exhibit a higher degree of compound complexity, combining several challenging aspects simultaneously, such as deep join chains, nested aggregations, and strict MySQL 8 constraints. Their inclusion serves to validate whether the observed performance trends remain stable under increased difficulty.

Each query in the test set is executed twice for every experimental configuration. This repetition is motivated by the stochastic nature of large language models, which may produce different outputs for the same input due to probabilistic decoding. In practice, it is observed that a query may be answered correctly in one run and incorrectly in another, even under identical conditions. Executing each query twice reduces the impact of this variance and provides a more robust estimate of model capability.

The number of repetitions is intentionally limited to two in order to balance statistical robustness with inference cost, reflecting realistic deployment scenarios where excessive retries would be economically undesirable.

### 3.4.2 Primary Metric: SQL Generation Success

The primary metric used to evaluate model performance is the SQL generation success rate. This metric measures the ability of a model to produce SQL queries that are not only syntactically valid, but also semantically and business-wise correct.

A generated SQL query is considered correct if and only if it satisfies all of the following criteria:

- Syntactic correctness: the query conforms to the MySQL 8 dialect and can be parsed and executed by the database engine without syntax errors.
- Execution correctness: the query executes successfully against the Northwind database and correctly implements the logical intent of the natural language

request, including appropriate joins, filters, and aggregations.

- Business correctness: the query respects domain-specific business rules implicit in the question, such as correct revenue computation formulas, proper handling of discounts, or appropriate aggregation levels.

This three-level definition of correctness is essential to exclude degenerate cases in which a query is syntactically valid but semantically incorrect or misleading from a business perspective.

For each query, a success is recorded if at least one of the two executions produces a correct SQL query according to the criteria above. The overall success rate is computed as the ratio between the number of successfully answered queries and the total number of queries in the test set. This definition reflects a system-level perspective, emphasizing the ability of the model to eventually produce a correct answer rather than penalizing isolated stochastic failures.

### 3.4.3 Efficiency Metrics and Orchestration Behavior

In addition to correctness, several secondary metrics are considered to characterize efficiency and operational behavior, which are critical factors in real-world deployments.

Inference latency is not explicitly measured in this evaluation. All experiments are conducted using the standard inference mode provided by OpenAI, which ensures low and relatively uniform response times across models. Since the primary objective of this work is to compare accuracy and cost-related trade-offs rather than absolute latency, and given the absence of batch inference or asynchronous execution, latency is assumed to be comparable across configurations.

Token cost per request is instead explicitly tracked and reported. This metric captures the average number of input and output tokens consumed during inference, accounting for prompt length, schema descriptions, few-shot examples, and follow-up interactions when applicable. Token cost is a critical indicator of economic efficiency, particularly when comparing contextual fine-tuning approaches—where large prompts must be supplied at inference time—with supervised fine-tuning strategies that encode knowledge directly into model parameters.

The primary object of evaluation in this study is the set of adaptation configurations described above. For configurations embedded in an orchestrated pipeline, a limited set of system-level indicators is additionally observed—including the number of retries triggered due to invalid or unsatisfactory outputs, and the number of follow-up interactions initiated by the orchestrator to refine intermediate results. However, the orchestrated pipeline as a whole is not subjected to a dedicated, deep quantitative evaluation (e.g., controlled ablations, end-to-end latency benchmarking, or statistically powered comparisons); instead, it is used mainly as an

integration setting to apply and qualitatively validate the selected configuration under realistic execution constraints.

### 3.4.4 Result Aggregation and Error Analysis

Results are aggregated at the query level rather than at the individual execution level. For each query, correctness is determined based on the success of at least one execution. This aggregation strategy avoids over-penalizing models for isolated decoding failures and aligns with realistic usage patterns where limited retries are acceptable.

For cost-related metrics, values are averaged across the two executions of each query and then averaged across the entire test set. This two-stage aggregation ensures that each query contributes equally to the final metrics, preventing queries with higher retry counts from disproportionately influencing the results.

In addition to aggregate success rates, errors are categorized into distinct failure classes, including syntactic errors, logical errors (e.g., incorrect joins or aggregation levels), and business-rule violations. This error taxonomy enables a finer-grained analysis of model weaknesses and supports qualitative comparisons between adaptation strategies such as contextual fine-tuning, supervised fine-tuning, and hybrid prompt-based approaches.

Overall, this evaluation protocol provides a comprehensive framework for assessing not only the accuracy of Text-to-SQL generation, but also the economic and operational implications of deploying different model configurations in real-world analytical systems.

## 3.5 Models and Experimental Configurations

### 3.5.1 Contextual Fine-Tuning Baseline

The first experimental configuration establishes a strong baseline based on contextual fine-tuning, in which all task-specific knowledge is injected at inference time through prompt engineering, without modifying the underlying model parameters. This configuration is designed to approximate the upper bound of what can be achieved using a prompt-only strategy.

The model employed in this baseline is GPT-4.1 Mini, used in its original, non-fine-tuned form. The choice of this model reflects a realistic deployment scenario in which a relatively efficient general-purpose LLM is adapted to a specialized task through context alone.

### System Prompt Design

The core of the contextual fine-tuning approach lies in the design of a highly structured system prompt, which encodes both domain knowledge and execution constraints. The prompt can be conceptually divided into four layers of injected information.

The first layer defines the **task role and operational scope** of the model. The model is explicitly instructed to act as an expert SQL assistant for the Northwind database using the MySQL 8 dialect. The output space is strictly constrained to a single `SELECT` statement, explicitly forbidding any form of data manipulation language (DML), data definition language (DDL), explanatory text, comments, or formatting artifacts. This restriction is essential to guarantee that the generated output can be executed directly by the database engine without post-processing.

The second layer introduces a set of hard syntactic constraints. These constraints explicitly prohibit unsupported SQL constructs and MySQL-incompatible features, such as ordered-set aggregates, `PERCENTILE_CONT`, `FILTER`, `QUALIFY`, full outer joins, user-defined variables, and inline assignments. Additionally, the prompt enforces strict aliasing rules, forbidding the reuse of newly defined aliases within the same `SELECT` scope and requiring explicit aliases for all subqueries and derived tables. These rules are designed to eliminate entire classes of syntactic errors that frequently arise in LLM-generated SQL.

The third layer encodes semantic constraints imposed by MySQL 8, with particular emphasis on the `ONLY_FULL_GROUP_BY` mode. The prompt explicitly states that every non-aggregated column in the `SELECT` clause must appear in the `GROUP BY`, and it forbids the application of window functions to raw columns in the presence of aggregation. Instead, a canonical two-stage pattern is prescribed: aggregation in an initial CTE, followed by window functions applied to the aggregated result. Similar canonical patterns are provided for other non-trivial operations, such as rolling statistics, year-over-year growth computation, and median estimation using ranking functions.

The fourth layer injects domain-specific business logic and schema conventions. This includes the canonical revenue formula, the correct interpretation of business terms such as "market" and "reseller", and explicit rules for handling geographic attributes. The prompt also enforces the use of exact table and column names as defined in the Northwind schema, preventing hallucinated schema elements and ensuring consistency with the actual database structure.

### Schema Injection and Prompt Size

In addition to the system-level rules, the complete schema of the Northwind database is provided to the model at inference time. This schema description includes table definitions, column names, and primary key-foreign key relationships.

Since the base model has not been exposed to the Northwind schema during pre-training, this explicit schema injection is critical to enable correct join reasoning and column selection.

As a consequence, each inference call in this configuration includes a large prompt, combining the schema description and the full set of SQL and business rules. While this significantly increases token consumption, it ensures that the model has access to all relevant constraints when generating the query.

#### Inference Configuration

Inference is performed using deterministic decoding with a temperature set to zero. This choice minimizes output variance and allows observed errors to be attributed to modeling limitations rather than stochastic sampling effects. All other decoding parameters are kept constant across experiments to ensure a fair comparison with supervised and hybrid approaches.

No few-shot examples are included in this baseline configuration. The model is required to generalize from declarative rules and schema descriptions alone, without being shown concrete examples of correct or incorrect SQL queries.

#### Role as Experimental Reference

This contextual fine-tuning configuration serves as the primary experimental baseline. It represents a best-case scenario for prompt-only adaptation, leveraging a powerful foundation model augmented with exhaustive contextual information. As such, it provides a reference point for evaluating the benefits of supervised fine-tuning and few-shot learning.

Importantly, this baseline also highlights a fundamental trade-off inherent to contextual fine-tuning approaches: while high levels of correctness can be achieved through extensive prompt engineering, this comes at the cost of increased inference-time token usage and reduced scalability. These limitations motivate the exploration of alternative configurations in which schema knowledge and rule compliance are partially or fully internalized through supervised fine-tuning.

### 3.5.2 Supervised Fine-Tuning: Initial Configuration (SFT-50)

The second experimental configuration introduces supervised fine-tuning (SFT) as a mechanism to internalize database-specific knowledge directly into the model parameters. In contrast to contextual fine-tuning, where all specialization is provided at inference time, SFT aims to encode recurrent patterns—such as schema structure, join relationships, and canonical aggregations—into the model itself.

This initial SFT configuration, referred to as SFT-50, represents the first attempt to specialize the model for the Northwind database using a limited but carefully curated training dataset.

### Training Dataset Construction

The fine-tuning dataset consists of 50 examples, each formatted as a conversation in `JSONL` according to OpenAI's fine-tuning specification. Each example is composed of three messages: a system message, a user message, and an assistant message.

The system message provides a concise but explicit definition of the task role and operational constraints. Unlike the extensive system prompt used in contextual fine-tuning, the system message in the SFT dataset is intentionally short. Its purpose is not to encode all SQL rules, but to define the expected output format, SQL dialect, and core business conventions. An example system message used across all samples is:

```
Sei un assistente SQL esperto per il database Northwind in MySQL.
Regole:  restituisci SOLO la query SQL finale; usa il dialetto MySQL;
solo SELECT;
usa solo tabelle/colonne esistenti; preferisci JOIN esplicite con
ON e chiavi esterne corrette;
se richiesto, calcola il fatturato come
quantity*unitPrice*(1-COALESCE(discount,0)).
```

The user message contains a natural language query expressed in Italian. These queries are designed to cover the full breadth of the Northwind schema, including customers, orders, products, categories, suppliers, and shipping information. The assistant message contains the corresponding SQL query written in valid MySQL 8 syntax.

The examples predominantly focus on structurally simple but schema-complete queries. Typical patterns include lifetime aggregations, top-$k$ rankings, and single-level `GROUP BY` queries. For instance, examples include identifying the customers with the highest number of orders, the most sold products by quantity, or computing revenue aggregated by shipping country.

The primary objective of this dataset is to teach the model:

- the exact names of tables and columns,
- the correct primary key-foreign key relationships,
- canonical join paths between entities,
- basic aggregation and ordering patterns.

Notably, the dataset does not emphasize complex SQL constructs such as window functions, multi-stage CTE pipelines, or edge cases related to MySQL 8 constraints.

### Fine-Tuning Procedure

The fine-tuning job is executed using an enterprise OpenAI API key, ensuring stable access to training resources and consistent model availability. The base model selected for fine-tuning is GPT-4.1 Mini, chosen to balance specialization potential with cost-efficiency.

Fine-tuning is performed using OpenAI's standard supervised fine-tuning pipeline with default hyperparameters. These defaults include a learning rate schedule, number of epochs, and batch size that are automatically selected based on dataset size and model architecture. The choice to use default hyperparameters is deliberate: given the relatively small dataset size, manual tuning of learning rate or epoch count would likely increase the risk of overfitting without addressing the core limitations of the training data.

During training, the model is optimized to maximize the likelihood of the assistant message conditioned on the system and user messages. This process effectively reinforces mappings from natural language queries to SQL patterns that are repeatedly observed in the dataset, allowing the model to internalize schema-specific knowledge.

### Inference Configuration and Prompting Strategy

At inference time, the fine-tuned model is invoked with a significantly reduced prompt compared to the contextual fine-tuning baseline. The full database schema and the extensive set of MySQL rules are no longer provided. Instead, the model relies primarily on the knowledge encoded during fine-tuning.

The inference prompt typically includes only a short system instruction consistent with the one used during training, followed by the user query. Decoding is performed deterministically with temperature set to zero, ensuring reproducible outputs and enabling direct comparison with other configurations.

No few-shot examples or additional rule descriptions are provided in this configuration. As a result, the model must generalize from the training data alone when faced with unseen queries.

### Observed Behavior and Limitations

Empirical evaluation of the SFT-50 configuration reveals a clear distinction between schema-level learning and procedural rule compliance. The model demonstrates strong performance on queries that resemble the structural patterns seen during training, particularly those involving simple joins and aggregations.

However, performance degrades significantly when the model is tested on complex analytical queries that require strict adherence to MySQL 8 constraints or multi-step reasoning patterns. The majority of observed errors are syntactic in nature,

often related to violations of `ONLY_FULL_GROUP_BY`, incorrect interactions between aggregation and window functions, or improper alias scoping.

These results indicate that while a small supervised dataset is sufficient to teach the model the structure of an unseen database, it is insufficient to override or refine the model's pre-existing SQL heuristics in edge cases governed by dialect-specific rules. This observation motivates the subsequent experiments aimed at extending the dataset, introducing rule-focused examples, and combining fine-tuning with inference-time guidance.

### 3.5.3 Incremental Fine-Tuning and Rule-Augmented Training

Following the limitations observed in the initial SFT-50 configuration, a second supervised fine-tuning strategy was explored with the goal of improving the model's ability to generate complex SQL queries that comply with strict MySQL 8 constraints. This strategy combines incremental fine-tuning with explicit rule exposure during training.

The underlying hypothesis of this experiment is that the errors observed in SFT-50—particularly those related to aggregation semantics, window functions, and alias scoping—could be mitigated by exposing the model to more complex examples and by reinforcing SQL rules directly during training.

#### Incremental Fine-Tuning Setup

Starting from the previously fine-tuned SFT-50 model, an incremental fine-tuning phase is performed using an additional set of 10 training examples. These examples are intentionally selected to exhibit significantly higher structural and semantic complexity compared to the original dataset.

In particular, the additional samples include:

- multi-stage query plans implemented via common table expressions (CTEs),
- combinations of aggregation and window functions,
- temporal groupings and derived metrics,
- non-trivial alias dependencies across query stages.

To further guide the model during training, the system message associated with these additional samples is extended to include a more detailed description of SQL rules and constraints, particularly those governing MySQL 8 behavior. This represents a deliberate attempt to internalize rule compliance directly into the model parameters rather than relying on inference-time prompting.

The incremental fine-tuning process uses the same OpenAI supervised fine-tuning pipeline and default hyperparameters as the initial SFT-50 experiment. No changes are made to learning rate schedules or epoch counts, as the small size of the incremental dataset makes hyperparameter tuning both unreliable and prone to overfitting.

### Empirical Results and Failure Analysis

Despite the increased complexity of the training examples and the richer system message, empirical evaluation shows that this incremental fine-tuning strategy does not lead to a measurable improvement in performance on the complex test queries. The success rate remains largely unchanged compared to the SFT-50 configuration and remains significantly below that of the contextual fine-tuning baseline.

A detailed analysis of the generated queries reveals that the majority of failures are not attributable to an inability to reason about query structure or database relationships. Instead, they are predominantly caused by violations of specific MySQL 8 syntactic and semantic constraints. Typical errors include:

- the application of window functions to raw columns in the presence of `GROUP BY`,
- missing or incorrect column exposure across CTE boundaries,
- illegal references to aliases outside their valid scope,
- violations of the `ONLY_FULL_GROUP_BY` constraint.

These observations suggest that the model's shortcomings are not due to insufficient expressivity or reasoning depth, but rather to a mismatch between its pre-trained SQL heuristics and the strict rules enforced by MySQL 8.

### Limitations of Rule Learning via Small-Scale SFT

The failure of incremental fine-tuning highlights a fundamental limitation of small-scale supervised fine-tuning for procedural rule learning. While the model successfully internalizes the Northwind schema with as few as 50 examples, learning or overriding SQL dialect-specific execution rules appears to be substantially more difficult.

This asymmetry can be explained by the model's pre-training history. The base model has already been exposed to a wide variety of SQL dialects and coding styles during pre-training. As a result, it possesses strong but generic SQL priors. Attempting to modify these priors using a small number of supervised examples is inherently challenging, particularly when the desired behavior requires consistent application of strict and sometimes counterintuitive constraints.

Furthermore, simply increasing the verbosity of the system message during training does not provide the model with actionable guidance on how to apply the rules. Without a sufficient number of concrete examples demonstrating correct behavior in edge cases, the model lacks the gradient signal necessary to adjust its internal representations.

#### Rationale Against Further Hyperparameter Tuning

At this stage, additional hyperparameter tuning—such as increasing the number of training epochs or adjusting the learning rate—was deliberately avoided. Given the limited size of the incremental dataset, such modifications would primarily amplify memorization effects rather than improve generalization. In practice, this would lead to overfitting on the few complex examples without addressing the underlying rule compliance problem.

This reasoning leads to the conclusion that the observed performance plateau is not a consequence of suboptimal training configuration, but rather of a structural mismatch between the learning signal provided by the dataset and the nature of the errors being corrected.

#### Implications for Subsequent Experiments

The results of this experiment demonstrate that supervised fine-tuning alone, when performed with limited data, is insufficient to reliably enforce strict SQL dialect rules in complex analytical queries. This insight motivates a shift in strategy toward hybrid approaches that combine the strengths of supervised fine-tuning for schema learning with inference-time mechanisms capable of explicitly grounding procedural rules.

In the following experiments, this insight leads to two complementary directions: first, the expansion and restructuring of the training dataset to explicitly encode rule-compliant patterns, and second, the integration of prompt-based techniques—most notably few-shot learning—to provide concrete examples of correct rule application at inference time.

### 3.5.4 Expanded Dataset and Rule-Centric Fine-Tuning (SFT-100)

Building on the limitations identified in the incremental fine-tuning experiment, a fourth experimental configuration is introduced: an expanded supervised fine-tuning setup explicitly designed to encode SQL rule compliance. This configuration, referred to as SFT-100, represents a deliberate shift from quantity-driven fine-tuning toward rule-centric data curation.

The primary objective of this experiment is to test whether increasing the volume of supervised examples—while simultaneously restructuring their content to emphasize procedural correctness—can mitigate the MySQL 8 compliance errors observed in previous configurations.

### Motivation and Experimental Hypothesis

The failure of incremental fine-tuning suggests that small-scale updates are insufficient to override the model's pre-trained SQL heuristics. In particular, violations related to aggregation semantics, alias scoping, and window function constraints persist even when complex examples and verbose system prompts are introduced.

This experiment is motivated by the hypothesis that:

> *SQL rule compliance can only be learned if rule-consistent execution patterns are repeatedly reinforced across a sufficiently large and diverse set of supervised examples.*

Rather than assuming that the model can generalize procedural rules from a few complex samples, SFT-100 explicitly targets the most common failure modes by embedding correct patterns directly into the training distribution.

### Dataset Expansion Strategy

The training dataset is expanded from 60 to 100 samples. However, the expansion is not uniform: additional examples are selectively designed to address the specific rule violations observed during evaluation of SFT-50 and incremental fine-tuning.

The newly introduced samples emphasize:

- correct separation of aggregation and window function stages using multi-layer CTEs,
- explicit column exposure across CTE boundaries,
- strict compliance with `ONLY_FULL_GROUP_BY` semantics,
- proper alias scoping and avoidance of illegal alias reuse,
- MySQL 8-compatible implementations of advanced analytics (e.g., rolling averages, year-over-year growth).

Each training example is constructed to demonstrate not only the final correct query, but also a canonical structural decomposition that the model can repeatedly observe and internalize.

### Rule-Centric System Prompt Design

In contrast to earlier fine-tuning stages, the system prompt used during SFT-100 is explicitly designed to act as a procedural specification rather than a high-level instruction.

The prompt enforces:

- strict output constraints (single `SELECT` statement, no comments or explanations),
- explicit prohibitions against unsupported SQL features and functions,
- detailed guidance on aggregation, window functions, and CTE usage,
- schema-level constraints specific to the Northwind database.

Crucially, this prompt is not merely verbose but operational: each rule is directly reflected in multiple supervised examples. The intent is to reduce the gap between abstract instructions and concrete execution patterns.

### Fine-Tuning Procedure

The fine-tuning process follows the same OpenAI supervised fine-tuning pipeline as previous experiments. The job is initialized using a corporate OpenAI API key, and all hyperparameters are left at their default values.

This choice is intentional. Given the increased dataset size and structural consistency of the samples, performance improvements—if any—are expected to arise from data quality and coverage rather than optimization-level adjustments.

Formally, the training objective remains unchanged:

$$\mathcal{L} = -\sum_{t} \log P(y_t \mid y_{<t}, x; \theta), \tag{3.1}$$

where $x$ represents the combined system and user prompt, and $y$ the target SQL query.

### Observed Outcomes

Evaluation results show a moderate improvement in syntactic correctness compared to SFT-50 and incremental fine-tuning. In particular, the frequency of outright invalid queries (i.e., queries rejected by the MySQL parser) decreases.

However, semantic correctness remains inconsistent for complex analytical queries. While the model more frequently adopts the correct high-level structure (e.g., introducing intermediate CTEs), subtle rule violations persist. Common failure cases include:

- incomplete column propagation between CTEs,
- misuse of aliases across query scopes,
- correct aggregation structure paired with incorrect grouping granularity.

These errors are less severe than in previous configurations but remain sufficient to cause execution failures or incorrect results.

#### Interpretation and Limitations

The results of SFT-100 indicate that expanding the dataset and embedding rule-centric examples yields diminishing returns beyond a certain point. Although the model demonstrates improved structural awareness, it still struggles to consistently apply procedural constraints across novel queries.

This suggests that SQL rule compliance—particularly for dialect-specific constraints such as those enforced by MySQL 8—is not easily reducible to pattern matching alone. Even with increased exposure, the model continues to rely on probabilistic heuristics rather than deterministic rule application.

From an experimental standpoint, this highlights a key limitation of supervised fine-tuning: while effective for learning schemas and common query templates, it is less suited for encoding strict execution semantics that must hold universally.

#### Consequences for Model Design

The SFT-100 experiment represents a critical inflection point in the experimental pipeline. The marginal gains achieved through dataset expansion do not justify further increases in supervised data volume without a corresponding change in methodology.

These findings motivate a transition toward hybrid approaches that combine:

- supervised fine-tuning for schema grounding and common query patterns,
- inference-time mechanisms (e.g., few-shot prompting) to enforce procedural correctness.

This transition is formalized in the next experimental configuration, which explicitly integrates rule-compliant examples at inference time rather than attempting to fully internalize them during training.

### 3.5.5 SFT-50 with Rules Injected at Inference Time

This configuration represents a hybrid adaptation strategy designed to isolate the effect of model capacity when rule-level constraints are provided exclusively at

inference time. The underlying model is the schema-specialized variant obtained through the SFT-50 procedure, trained on 50 schema-covering examples, without any additional fine-tuning stages.

No further supervised updates are applied to the model parameters. Instead, all MySQL 8 syntactic and semantic constraints are injected at inference time via an extended system prompt. The prompt explicitly enumerates the relevant dialect rules (e.g., strict grouping semantics, alias usage constraints, and aggregation requirements), but does not include any few-shot examples demonstrating their application. As a result, the model is required to reconcile these explicit instructions with its pre-trained and fine-tuned parametric priors without procedural guidance.

To assess whether residual errors observed in previous configurations were attributable to insufficient model capacity rather than to structural resistance to rule adoption, this setup was evaluated using two different base models:

- GPT-4.1 Mini;
- GPT-4.1 (full).

The empirical results show a near-identical performance between the two models. GPT-4.1 Mini correctly generated 52 out of 60 queries, while GPT-4.1 achieved 53 out of 60 correct generations. The marginal improvement obtained with the larger model is statistically negligible and does not alter the qualitative failure modes observed in the outputs.

This outcome is particularly informative because it isolates model capacity as an experimental variable while holding the training data, prompt structure, and inference-time constraints constant. The similarity in performance strongly suggests that the primary limitation is not insufficient reasoning capability or representational power. Instead, the bottleneck arises from the conflict between explicit rule instructions provided at inference time and entrenched parametric priors learned during large-scale pre-training.

From a knowledge-editing perspective, this configuration highlights the limited effectiveness of declarative rule injection when such rules contradict high-frequency patterns embedded in the model's weights. Even when the rules are fully specified and the base model is upgraded, the absence of procedural demonstrations prevents reliable override of these priors. This finding aligns with the broader literature on resistance to editing, where increasing model size alone does not guarantee improved controllability when updates compete with consolidated knowledge.

Finally, from a practical standpoint, the negligible accuracy gain obtained with GPT-4.1 does not justify its substantially higher cost, both in training and inference. Consequently, this configuration provides strong empirical evidence that upgrading model capacity is not an efficient solution for enforcing dialect-specific constraints, reinforcing the motivation for hybrid strategies that combine schema-level fine-tuning with inference-time procedural guidance.

### 3.5.6 Hybrid Configuration: SFT with Prompt-Time Few-Shot Conditioning

This subsection describes the final hybrid configuration evaluated in this work, which combines supervised fine-tuning (SFT) with prompt-time rule conditioning and few-shot exemplars. This setup, referred to as *SFT + prompt + few-shot*, represents the most advanced standalone Text-to-SQL configuration evaluated prior to the introduction of the orchestrated pipeline discussed in Section 3.6.

The motivation for introducing this configuration emerges directly from the limitations observed in the preceding experiments. Supervised fine-tuning alone proved effective at internalizing database-specific structural knowledge (tables, columns, primary keys, foreign keys), but insufficient to reliably enforce strict MySQL 8 dialect constraints. In particular, errors related to `ONLY_FULL_GROUP_BY`, window functions applied to non-aggregated columns, alias scoping, and improper reuse of intermediate expressions persisted even after incremental fine-tuning.

#### Motivation for Prompt-Time Few-Shot Conditioning

The introduction of few-shot exemplars at inference time is grounded in the distinction between parametric knowledge acquisition and procedural constraint enforcement. In our experiments, SFT efficiently internalizes database-specific knowledge (schema elements and canonical join paths), but shows clear limitations when the objective is to enforce dialect-level procedural constraints that conflict with the model's pre-trained SQL priors.

This behavior is consistent with the resistance to rule updates" phenomenon described in Section 2.2.2: when the desired output distribution contradicts high-frequency patterns learned during pre-training, a small supervised dataset provides an insufficient gradient signal to reliably override entrenched heuristics. In practice, this yields a model that "knows" the schema but still produces SQL that violates MySQL 8 constraints in edge cases.

Few-shot prompting addresses this limitation by steering model behavior at decoding time rather than attempting to modify the underlying parameters. By providing explicit demonstrations of incorrect versus correct SQL constructions, the prompt supplies procedural guidance that is immediately actionable during generation, reducing reliance on uncertain parameter-level updates.

#### Inference Prompt Structure

In this configuration, inference is performed using a single, fixed prompt template composed of three components:

1. A system instruction defining the model as an expert SQL assistant for the Northwind MySQL 8 database.

2. An explicit set of syntactic and semantic constraints (e.g., aggregation requirements, window function limitations, aliasing constraints, and MySQL 8–specific restrictions).

3. A small set of few-shot exemplars targeting the most frequent failure modes observed in earlier experiments.

The exemplars are embedded directly within the prompt and precede the user query. Each exemplar targets a distinct and recurring error category, such as the misuse of window functions in conjunction with `GROUP BY`, improper alias reuse across query scopes, or violations of strict aggregation rules.

Crucially, the exemplars are procedural rather than task-specific: they do not solve concrete user queries, but instead demonstrate how specific SQL rules must be applied in practice. This design choice directly responds to the limitation of SFT highlighted in Section 2.2.2: rule compliance is easier to induce when the model is shown concrete, contrastive patterns of correct behavior.

The user's natural-language question is appended to the prompt as a single input, after which the model is required to generate exactly one SQL query, without explanations, comments, or intermediate reasoning steps.

### Few-Shot Exemplars: Scope and Consistency

The set of few-shot exemplars is fixed and identical across all inference calls. The examples are not dynamically selected, not query-dependent, and are not used during the supervised fine-tuning phase. Their sole purpose is to enforce procedural compliance at inference time.

The number of exemplars is deliberately kept small (approximately one per major error category) to balance the need for procedural guidance with inference-time token cost. This trade-off is central to the overall motivation for hybrid adaptation: by storing schema knowledge parametrically (via SFT) and procedural rule enforcement in-context (via exemplars), the system reduces prompt size compared to a fully contextual baseline while retaining high correctness.

### Model and Decoding Configuration

The underlying language model used in this configuration is the same supervised fine-tuned model described in Sections 3.5.2 through 3.5.4. No additional fine-tuning or parameter updates are performed. Inference is conducted using deterministic decoding with temperature set to zero, ensuring consistent and reproducible outputs and enabling a reliable comparison across experimental conditions.

#### Rationale for Comparison with Contextual Fine-Tuning

This hybrid configuration is designed to enable a fair comparison with a contextual fine-tuning approach augmented with few-shot prompting. While contextual fine-tuning relies on injecting the full database schema and rules into the prompt at inference time, the SFT-based approach internalizes schema knowledge parametrically and relies on prompt-time few-shot exemplars for procedural constraint enforcement.

Comparing these two configurations under equivalent few-shot conditions isolates the impact of where knowledge is stored—in model parameters versus inference context—and how this choice affects both accuracy and inference cost. The quantitative results of this comparison are presented and analyzed in the following Results section.

### 3.5.7 Technical Details of Supervised Fine-Tuning

This subsection provides a detailed account of the supervised fine-tuning (SFT) procedure adopted in the experimental study. The objective is to ensure full reproducibility and to clarify the design choices that influenced the observed model behavior.

#### Base Model and Fine-Tuning Objective

All supervised fine-tuning experiments start from the `gpt-4.1-mini` base model. The objective follows the standard supervised fine-tuning setup for decoder-only LLMs: given a sequence of `system`, `user`, and `assistant` messages, training maximizes the likelihood of the assistant SQL conditioned on the preceding messages.

#### Dataset Composition and Encoding Format

The fine-tuning dataset was constructed from the Northwind database and encoded in the `jsonl` format required by the OpenAI fine-tuning API. Each training example consists of a full conversational trace, including:

- a `system` message providing high-level task context,
- a `user` message containing a natural language query in Italian,
- an `assistant` message containing the corresponding SQL query in MySQL 8 dialect.

The initial dataset used for SFT contained 50 examples. These samples were designed to provide broad schema coverage rather than query complexity, ensuring

that all major tables, columns, primary keys, and foreign key relationships of the Northwind database were represented. The focus at this stage was on internalizing the relational structure of the database rather than enforcing advanced SQL constraints.

A representative training sample is shown below:

```
{"messages": [
  {"role": "system", "content": "Sei un assistente SQL esperto per
   il database Northwind in MySQL.\nRegole: restituisci SOLO la
   query SQL finale; usa il dialetto MySQL; solo SELECT;\nusa solo
   tabelle/colonne esistenti; preferisci JOIN esplicite con ON e
   chiavi esterne corrette;\nse richiesto, calcola il fatturato
   come quantity*unitPrice*(1-COALESCE(discount,0))."},
  {"role": "user", "content": "Elenca i 10 clienti con più ordini
   (lifetime)"},
  {"role": "assistant", "content": "SELECT c.CompanyName, COUNT(*)
   AS ordersCount\nFROM Orders o\nJOIN Customers c ON o.CustomerID
   = c.CustomerID\nGROUP BY c.CustomerID, c.CompanyName\nORDER BY
   ordersCount DESC\nLIMIT 10;"}
]}
```

All assistant responses consist exclusively of executable SQL queries. No explanations, comments, or auxiliary text are included, ensuring that the model learns a strict mapping from natural language to SQL output.

### System Message Design in Training

The system message used during fine-tuning is intentionally concise: it defines the task role and a minimal set of output constraints, but it does not attempt to encode the full set of MySQL 8 edge-case rules. This choice is consistent with the analysis in Section 2.2.2: rule compliance is difficult to imprint parametrically without many targeted examples, whereas schema knowledge can be internalized more effectively.

### Fine-Tuning Hyperparameters and Training Configuration

All fine-tuning jobs were executed using the OpenAI fine-tuning infrastructure with default hyperparameters. No manual tuning of optimization parameters was performed. This choice reflects a realistic deployment scenario and avoids overfitting risks associated with aggressive hyperparameter manipulation on small datasets.

The configuration can be summarized as follows:

- Learning rate: default (OpenAI-managed schedule)
- Number of epochs: default
- Batch size: automatically selected by the platform
- Optimizer: OpenAI-managed (not externally configurable)
- Random seed: platform-defined
- Temperature during inference: 0 (deterministic decoding)

The decision not to adjust learning rate, epoch count, or batch size was deliberate. Given the limited number of training samples, increasing training intensity would primarily amplify memorization effects without addressing the underlying issue of rule-level knowledge resistance discussed in Section 2.

**Fine-Tuning Execution**

Fine-tuning jobs were launched using the OpenAI fine-tuning infrastructure. Once a job completed, the resulting fine-tuned model identifier was used directly for inference in subsequent experiments without further modification, keeping decoding settings identical across configurations.

## 3.6 Implementation of Langgraph Pipeline

### 3.6.1 Orchestrator–Worker Architecture and Decomposition Strategy

The proposed pipeline adopts an explicit Orchestrator–Worker–Orchestrator architectural pattern, designed to separate global reasoning and planning from local, constraint-bound Text-to-SQL generation. This separation is central to the robustness and interpretability of the system, as it allows different models to be specialized for distinct cognitive roles while maintaining a clear and controllable execution flow.

At a high level, the orchestrator is responsible for understanding the user's analytical intent and transforming a potentially complex natural language request into a structured execution plan. The worker models, in contrast, are exclusively tasked with translating individual, well-scoped sub-questions into executable SQL queries. The orchestrator re-enters the loop only after execution, in order to validate completeness and synthesize results.

The orchestrator is implemented using a general-purpose language model (GPT-4.1 Mini) and operates in a purely coordinative capacity. It never generates

SQL directly, does not execute queries, and does not access the database at runtime. Its only source of database knowledge is the schema, which is injected into the prompt as static context. This design allows the orchestrator to reason about tables, relationships, and plausible aggregations, while remaining completely decoupled from the actual data. As a result, the planning phase is data-agnostic and reproducible, and does not depend on intermediate query results.

Upon receiving a user question, the orchestrator performs an initial decomposition step. The goal of this step is to break down the original request into a set of atomic sub-questions, each of which can be answered by a single SQL query. Sub-questions are phrased in natural language and are intentionally narrow in scope, typically corresponding to one aggregation, one grouping dimension, or one specific analytical slice. Alongside each sub-question, the orchestrator produces a short description of the expected output, specifying the relevant columns and the type of aggregation involved. When applicable, the orchestrator also encodes logical dependencies that indicate how sub-results should be interpreted or ordered during the final analysis phase.

The output of the decomposition step is serialized into a structured JSON format, which is explicitly enforced through prompt instructions and validated during parsing. Each sub-task is represented as an object containing an identifier, the natural language sub-question, a short description of its analytical role, and an optional list of conceptual dependencies. These dependencies are semantic rather than technical: they do not imply that the execution of one sub-query provides input to another, but instead guide the orchestrator during result interpretation and synthesis.

A key design constraint of the decomposition strategy is the enforcement of an upper bound on the number of generated sub-queries. The orchestrator is instructed to produce no more than $N = 8$ sub-tasks. This limit is motivated by practical considerations of latency, cost, and cognitive tractability. Empirically, most analytical questions in the target domain can be resolved with three to five sub-queries, while larger decompositions tend to introduce redundancy or overly fine-grained analyses. The constraint is implemented both at the prompt level, by explicitly instructing the model to cap the number of sub-questions, and at the application level, by truncating any excess sub-tasks after parsing. This dual enforcement ensures that the pipeline remains stable even in cases where the model deviates from the expected output format.

Once the decomposition plan is generated, execution is delegated to the worker models. Workers are instances of a supervised fine-tuned Text-to-SQL model specialized for the Northwind schema and MySQL 8 dialect. Each worker invocation handles exactly one sub-question and is completely stateless. Workers do not receive the original user query, the decomposition plan, or the results of other sub-queries. This isolation is intentional: it enables parallel execution, simplifies error attribution,

and prevents unintended information leakage between sub-tasks.

The only shared context across worker invocations is the static system prompt, which encodes the database schema and the syntactic and semantic constraints of the SQL dialect. In some experimental configurations, this prompt may also include few-shot examples, but the operational interface remains unchanged. Because workers are stateless, failures or retries associated with one sub-query do not affect the execution of others, and the system can selectively re-invoke only the components that require correction.

Overall, this architectural design clearly delineates responsibilities within the pipeline. The orchestrator performs high-level reasoning, decomposition, and later validation, while the workers perform localized, schema-constrained SQL generation. By making decomposition explicit and bounded, and by enforcing stateless execution at the worker level, the pipeline achieves a balance between flexibility, robustness, and computational efficiency in handling complex analytical queries.

### 3.6.2 Parallel Execution, Error Handling, and Dynamic Guardrails

After decomposition, the pipeline executes sub-queries with three design goals: (i) parallelism, (ii) localized failures, and (iii) feedback-driven correction.

#### Parallel dispatch

Each sub-question is dispatched independently to a worker. Since workers are stateless and sub-queries are typically independent, execution can be parallelized (e.g., via a `ThreadPoolExecutor`), reducing end-to-end latency for requests that decompose into multiple aggregations or analytical slices.

#### Per-subtask execution steps

For each sub-task, the pipeline follows a fixed sequence:

1. SQL generation: invoke the worker with the static system prompt and the sub-question.
2. Lightweight validation: reject obviously invalid outputs (e.g., non-`SELECT` statements, missing terminator, forbidden constructs).
3. Database execution: run the SQL against MySQL 8.
4. Outcome capture: record either (a) a result set or (b) an execution error (e.g., missing columns, alias scope errors, `ONLY_FULL_GROUP_BY` violations).

#### Retries with execution feedback

If validation or execution fails, a bounded retry loop is triggered:

1. capture the database error message,
2. inject it as structured feedback together with the original sub-question,
3. re-invoke the worker to produce a corrected SQL query.

Retries are scoped strictly to the failing sub-task and limited to a small number of attempts (e.g., two retries) to keep costs predictable and avoid pathological loops.

#### Dynamic guardrails and pipeline state

Dynamic guardrails are implemented by surfacing execution-time errors to the worker as actionable signals, effectively turning database feedback into additional inference-time supervision.

All outcomes (queries, results, retries, and terminal failures) are recorded in a global LangGraph state. For each sub-task, the state stores:

- generated SQL queries,
- execution status,
- result set (if available),
- number of retries.

This explicit state representation allows the orchestrator to reason not only about the returned data but also about execution reliability and completeness.

### 3.6.3 Completeness Check, Follow-Up Loop, and Final Synthesis

Once the initial set of sub-queries has been executed and their results collected, the pipeline enters the final reasoning phase, which is again coordinated by the orchestrator model. This phase is responsible for deciding whether the available information is sufficient to answer the original user question and, if not, for triggering additional targeted investigations.

### Completeness Check

The first step is a completeness check performed by the orchestrator. At this point, the model is provided with:

- the original user question;
- the list of executed sub-questions;
- the corresponding result sets, pre-processed into a readable tabular form.

Based on this input, the orchestrator evaluates whether the results collectively cover all aspects of the original analytical intent. The outcome is a binary decision: either the results are deemed sufficient, or additional information is required.

### Criteria for Follow-Up Queries

If the completeness check fails, the orchestrator generates one or more follow-up sub-questions driven by criteria observed in the intermediate results, such as:

- anomalous values or outliers that warrant deeper inspection;
- partial coverage of the requested dimensions (e.g., missing segments, time ranges, or categories);
- empty or unexpectedly small result sets, suggesting overly restrictive filters or incorrect assumptions;
- emergent patterns that appear relevant but were not explicitly requested in the initial decomposition.

Each follow-up is formulated in the same atomic, Text-to-SQL-friendly manner as the initial sub-questions and is marked in the pipeline state as a follow-up task. Follow-up queries are dispatched to the worker models and executed using the same parallel and error-handling mechanisms described earlier.

### Iterative Loop and Termination Condition

The pipeline supports an iterative follow-up loop, but the number of iterations is strictly bounded (in practice, two to three cycles) to prevent unbounded exploration and excessive inference costs.

Termination occurs when either:

- the orchestrator judges the accumulated results sufficient to answer the original question, or
- the maximum number of iterations is reached, in which case the system proceeds with the best available information.

#### Handling of Anomalies and Incompleteness

Rather than silently ignoring problematic outputs, the orchestrator incorporates anomalies and incomplete evidence into its reasoning. Empty result sets, inconsistent aggregates, or partially successful executions are explicitly acknowledged and can influence both the generation of follow-up queries and the framing of the final answer.

#### Final Synthesis and Cognitive Role of the Orchestrator

Once the iterative loop terminates, the orchestrator synthesizes the final answer from the complete collection of result sets (initial and follow-up). The final output integrates:

- quantitative findings derived from the executed SQL queries;
- qualitative assessments of trends, differences, or anomalies;
- explicit acknowledgment of limitations or uncertainties arising from incomplete data.

The system is considered "finished" when the orchestrator judges that a semantically adequate and well-supported answer can be produced, emphasizing reasoning completeness over procedural completeness.

## 3.7 Quantitative Results

This section presents the quantitative results obtained from the experimental evaluation of the different text-to-SQL approaches described in the previous sections. The goal is to compare the effectiveness of contextual fine-tuning, supervised fine-tuning, and hybrid approaches combining fine-tuning with prompting and few-shot learning, both in terms of accuracy and robustness across increasingly complex queries.

### 3.7.1 Overall Performance Across Configurations

Table 3.1 reports the aggregated results for all evaluated configurations. Each configuration was tested on a set of 30 complex queries, with each query executed twice, resulting in a total of 60 runs per configuration. For the final comparison between the best-performing approaches, the test set was extended to 40 queries, corresponding to 80 total runs.

These results immediately highlight substantial differences in performance between the various approaches. Pure supervised fine-tuning with a limited number

**Table 3.1:** Aggregated success counts and accuracy across experimental configurations

| Configuration | Successful Runs | Total Runs | Accuracy (%) |
|---|---|---|---|
| Contextual FT (schema in prompt) | 51 | 60 | 85.0 |
| SFT (50 samples) | 24 | 60 | 40.0 |
| SFT (50 + 10 incremental samples) | 28 | 60 | 46.7 |
| SFT (100 samples) | 38 | 60 | 63.3 |
| SFT (50 samples + prompting rules) | 52 | 60 | 86.7 |
| SFT (50 samples + prompting + few-shot) | 58 | 60 | 96.7 |
| Contextual FT (+ few-shot) | 58 | 60 | 96.7 |
| SFT (50 + prompting + few-shot), extended test | 77 | 80 | 96.3 |
| Contextual FT (+ few-shot), extended test | 75 | 80 | 93.8 |

of samples performs significantly worse than the contextual fine-tuning baseline, confirming that a small SFT dataset is insufficient to generalize to complex SQL queries. Incremental fine-tuning with additional complex examples leads only to marginal improvements, suggesting that the observed errors are not primarily due to query complexity alone, but rather to systematic issues related to SQL dialect constraints and syntactic rules.

Increasing the size of the SFT dataset from 50 to 100 samples results in a notable improvement, yet performance remains well below that of contextual fine-tuning. This indicates that while additional training data helps, the cost of collecting and curating sufficiently large and representative datasets quickly becomes prohibitive.

A major performance shift is observed when prompting rules are introduced at inference time on top of a model fine-tuned on 50 samples. In this configuration, performance slightly surpasses the contextual fine-tuning baseline. This result suggests that supervised fine-tuning is particularly effective at teaching the model the database schema and relational structure, while explicit prompting remains necessary to enforce strict SQL dialect rules.

The best overall performance on the 30-query test set is achieved by combining supervised fine-tuning with prompting and few-shot examples, reaching 58 successful runs out of 60. Applying the same few-shot strategy to the contextual fine-tuning baseline yields identical accuracy. However, on the extended test set of 40 highly complex queries, the SFT-based configuration performs better (77 successful runs out of 80) than the contextual fine-tuning baseline (75 out of 80), indicating a small but consistent advantage for the hybrid SFT approach under the most challenging conditions.

### 3.7.2 Analysis by Experimental Phase

The results reveal a clear progression across experimental phases. Early SFT configurations fail primarily due to syntactic errors and violations of MySQL-specific constraints, which become more frequent as query complexity increases. These errors persist even when complex examples are added incrementally, indicating that a small number of demonstrations is insufficient to override the model's prior assumptions about SQL syntax.

The introduction of explicit prompting rules at inference time significantly reduces these errors by providing a structured constraint mechanism. However, prompting alone does not fully resolve all failure modes, as the model still lacks concrete demonstrations of how the rules should be applied in practice. Few-shot examples fill this gap by providing executable patterns that the model can directly imitate, resulting in a sharp increase in success rate.

Importantly, the equivalence between SFT plus few-shot and contextual FT plus few-shot demonstrates that the decisive factor for peak performance is not the fine-tuning strategy itself, but the availability of explicit, high-quality demonstrations at inference time. The key difference between the two approaches lies not in accuracy, but in inference cost and scalability, as discussed in subsequent sections.

### 3.7.3 Summary of Quantitative Findings

Overall, the quantitative results demonstrate that supervised fine-tuning alone is insufficient for complex text-to-SQL tasks under realistic constraints. However, when used as a mechanism to internalize database structure and combined with lightweight prompting and few-shot learning, SFT enables performance comparable to contextual fine-tuning while shifting part of the required knowledge from the prompt into the model parameters. In practice, this can reduce prompt length at inference time, at the cost of additional training and model-management overhead. These results motivate the architectural choices described in the following sections and justify the integration of the fine-tuned model into a modular, orchestrated pipeline.

## 3.8 Cost Analysis and Model Selection Rationale

This section analyzes the computational cost of the evaluated configurations and relates it to their empirical performance, with the goal of motivating the final architectural and modeling choices. The analysis focuses on the effective costs observed in the experimental campaign.

Table 3.2 summarizes, for each configuration, (i) the number of inference API calls performed in evaluation, (ii) the corresponding inference cost, (iii) the one-time

**Table 3.2:** Training, inference, and total cost for each experimental configuration.

| Configuration | API Calls | Inference Cost ($) | Training Cost ($) | Total Cost ($) |
|---|---|---|---|---|
| Contextual FT (schema in prompt) | 60 | 0.1646 | 0.0000 | 0.16 |
| SFT (50 samples) | 60 | 0.0951 | 0.0124 | 0.11 |
| SFT (50 + 10 incremental samples) | 60 | 0.0951 | 0.0149 | 0.11 |
| SFT (100 samples) | 60 | 0.0951 | 0.0248 | 0.12 |
| SFT (50 samples + prompting rules) | 60 | 0.2005 | 0.0124 | 0.21 |
| SFT (50 samples + prompting rules) – GPT-4.1 | 60 | 1.1059 | 0.0828 | 1.19 |
| SFT (50 samples + prompting + few-shot) | 80 | 0.3180 | 0.0124 | 0.33 |
| Contextual FT (+ few-shot) | 80 | 0.4055 | 0.0000 | 0.41 |

supervised fine-tuning cost (when applicable), and (iv) the resulting total. These values allow a direct comparison of how different adaptation strategies shift cost between training time and inference time.

### 3.8.1 Experimental Cost Scope

All reported experimental results are based on a limited and controlled number of API calls. For each configuration, the evaluation consisted of 30 test queries executed over two independent runs, for a total of 60 inference calls. The final comparison between the two best-performing configurations (SFT with rules and few-shot versus contextual prompting with few-shot) was conducted on a larger test set of 40 queries, again evaluated over two runs, resulting in 80 inference calls per configuration.

Under this setup, the total cost of the entire experimental campaign remains modest. The cumulative inference cost across all configurations is approximately $2.48, while the total cost associated with supervised fine-tuning jobs is approximately $0.15. The overall cost of the full experimental evaluation is therefore around $2.63. These values demonstrate that, at experimental scale, cost differences are primarily relevant in relative terms rather than absolute ones.

### 3.8.2 Model Capacity versus Cost: GPT-4.1 Mini vs GPT-4.1

A key objective of the cost analysis is to assess whether higher model capacity yields commensurate gains in Text-to-SQL performance. This question is directly addressed by the configuration in which the same SFT-50 schema-specialized model is used with rules injected at inference time and evaluated using two base models: GPT-4.1 Mini and GPT-4.1.

The observed success rates are nearly identical, with GPT-4.1 Mini correctly generating 52 out of 60 queries and GPT-4.1 generating 53 out of 60. This marginal improvement does not alter the qualitative error patterns and is not statistically

meaningful in practice. In contrast, the cost difference is substantial: inference with GPT-4.1 is approximately 5.5 times more expensive than with GPT-4.1 Mini, and training costs are also significantly higher.

These results indicate that the dominant source of errors is not insufficient reasoning capacity, but rather structural limitations related to the interaction between inference-time rules and the model's parametric priors. Increasing model size alone does not resolve this issue. Consequently, GPT-4.1 Mini represents the preferred choice for this task, as it achieves comparable accuracy at a fraction of the cost.

### 3.8.3 Effect of Fine-Tuning on Cost and Accuracy

Pure supervised fine-tuning configurations (SFT-50, SFT-50+10, and SFT-100) exhibit lower inference costs due to shorter prompts, but this reduction comes at the expense of a substantial drop in accuracy. Even the strongest of these configurations fails to exceed a success rate of approximately 63%, which is insufficient for reliable analytical querying. The reduction in inference cost, while measurable, does not compensate for the loss in correctness and robustness.

Hybrid configurations that combine schema-level fine-tuning with inference-time constraints recover a significant portion of the lost accuracy. However, the experiment shows that rule injection without procedural examples remains brittle, reinforcing the need for either few-shot guidance or fully contextual prompting.

### 3.8.4 Final Configuration Comparison

The two best-performing configurations in terms of accuracy are:

- SFT-50 with extended rules and few-shot examples at inference time;
- Contextual prompting with extended rules and few-shot examples, without any fine-tuning.

Both configurations achieve the same success rate of 58 correct queries out of 60 (96.7%) on the 30-query test set. On the extended comparison with 40 highly complex queries (two runs, 80 total calls), the SFT-based configuration performs slightly better than contextual fine-tuning (77/80 versus 75/80).

At the experimental scale, absolute costs remain modest, but Table 3.2 shows a clear difference in cost allocation and total. The SFT-based configuration is cheaper overall in the extended setting ($0.33 total, consisting of $0.318 inference and $0.012 training), because schema knowledge is stored parametrically and prompts can be kept shorter at inference time. In contrast, the contextual configuration incurs a higher total cost ($0.41), driven entirely by higher inference cost due to longer prompts, while avoiding any training overhead and simplifying iteration.

From an operational perspective, the contextual approach also offers greater flexibility. Prompt updates allow rapid iteration on rules and constraints without retraining, whereas the fine-tuned configuration requires managing and redeploying a specialized model. Given identical performance, lower cost, and higher maintainability, the contextual configuration with few-shot examples is therefore preferable.

### 3.8.5 Summary

Overall, the cost analysis supports three main conclusions. First, increasing model capacity beyond GPT-4.1 Mini does not yield proportional gains for Text-to-SQL generation under the examined constraints. Second, pure supervised fine-tuning is not cost-effective due to its negative impact on accuracy. Third, when accuracy is held constant, hybrid and contextual approaches are both viable, but they differ in where they allocate cost and operational complexity: contextual prompting with few-shot examples avoids any training cycle and simplifies iteration (prompt-only changes), whereas SFT-based alternatives require training and model management but can reduce inference-time prompt length and, in the extended setting considered here, the overall cost. These considerations motivate selecting the contextual few-shot configuration as the final system design, prioritizing maintainability given comparable accuracy.

# Chapter 4

# Conclusion

## 4.1 Summary and Key Findings

This thesis investigated the application of large language models to the problem of natural language to SQL translation over an enterprise relational database, with particular focus on schema specialization, rule compliance, and pipeline-level orchestration. The work evaluated multiple adaptation strategies, including contextual fine-tuning, supervised fine-tuning (SFT), and hybrid approaches combining parametric specialization with inference-time rule injection.

The experimental results demonstrated that schema specialization is a critical factor in achieving high translation accuracy. Models that were explicitly exposed to the database schema, either through contextual conditioning or supervised fine-tuning, significantly outperformed baseline configurations lacking schema grounding. In particular, supervised fine-tuning enabled the model to internalize relational structure and produce syntactically and semantically valid queries with high reliability.

A key finding of this work is that rule compliance, especially for strict SQL dialect constraints such as MySQL 8 grouping semantics, is more effectively enforced through inference-time conditioning rather than relying exclusively on parametric learning. Hybrid configurations combining schema-specialized SFT models with explicit rule injection at inference time achieved the highest overall accuracy, reaching up to 96.7% success rate on the evaluation set. This confirms the distinction between parametric knowledge acquisition and behavioral control at inference time, and highlights the importance of combining both mechanisms.

The comparative evaluation between GPT-4.1 Mini and GPT-4.1 further demonstrated that model scale and general cognitive capability were not the primary limiting factors. Both models achieved nearly identical performance when provided with equivalent schema specialization and inference-time constraints. This

result indicates that failures in rule compliance were primarily due to conflicts between entrenched parametric priors and newly introduced constraints, rather than insufficient reasoning ability.

In addition to model-level improvements, this thesis introduced an orchestrated pipeline architecture based on an Orchestrator–Worker pattern implemented with LangGraph. This architecture separates high-level reasoning, decomposition, and synthesis from specialized SQL generation, enabling modular execution, parallelism, and structured error handling. The pipeline demonstrated the ability to decompose complex analytical queries into atomic subtasks and to integrate their results into coherent final responses.

Overall, the results confirm that reliable natural language access to relational databases can be achieved by combining schema-specialized fine-tuning, inference-time constraint injection, and structured orchestration. This approach provides a practical and scalable foundation for deploying large language models in enterprise data environments.

## 4.2 Limitations and Future Work

Despite the promising results, this work presents several limitations that highlight important directions for future research.

A primary limitation concerns the size of the fine-tuning datasets. Due to cost constraints and the lack of publicly available datasets specifically tailored to the target database schema, supervised fine-tuning was performed using relatively small datasets, typically consisting of tens of samples rather than hundreds or thousands. While these datasets were sufficient to demonstrate clear improvements in schema specialization, larger training sets would likely enable more robust parametric internalization of both schema structure and dialect-specific constraints. Prior research suggests that the full benefits of supervised fine-tuning often emerge when models are exposed to several hundred or more high-quality examples. Future work should therefore explore larger-scale fine-tuning with expanded datasets to better characterize scaling effects and convergence behavior.

A second limitation relates to the nature of the database used in the experiments. The evaluation was conducted on a structured but relatively small and synthetic database rather than a real enterprise-scale dataset containing millions or billions of rows. As a result, it was not possible to fully stress-test the orchestrated pipeline under realistic large-scale conditions. The pipeline architecture was specifically designed to address scalability challenges by decomposing complex queries into smaller subtasks and enabling distributed execution. However, without access to a truly large dataset, it was not possible to extensively evaluate performance under heavy load, large result sets, or realistic latency constraints. Future work should

validate the pipeline on real-world enterprise databases to assess its scalability, efficiency, and robustness in production environments.

Another limitation concerns the termination mechanism of the orchestrated pipeline. The iterative follow-up process, in which the orchestrator evaluates completeness and generates additional subtasks if needed, is currently governed by a fixed maximum number of iterations. This parameter acts as a safeguard against infinite loops but does not allow the model to autonomously determine when sufficient information has been collected to produce a complete and reliable answer. Ideally, the orchestrator should be able to dynamically assess convergence based on semantic completeness, consistency of results, and absence of anomalies, rather than relying on externally imposed iteration limits. Developing adaptive termination criteria driven by model confidence or convergence signals represents an important direction for future work.

Future research could also explore more advanced orchestration strategies, including adaptive decomposition based on query complexity, dynamic allocation of computational resources, and integration with database query planners. Additionally, further investigation into hybrid adaptation strategies combining fine-tuning, retrieval augmentation, and structured prompting could improve both robustness and efficiency.

Finally, evaluating these techniques in real enterprise settings, with larger schemas, heterogeneous data sources, and evolving database structures, would provide a more comprehensive understanding of their practical applicability. Such studies would help establish best practices for deploying large language model-based query systems in production environments.

# Bibliography


[1] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165 (cit. on pp. 14, 15, 31, 32).

[2] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. «Improving language understanding by generative pre-training». In: (2018) (cit. on p. 17).

[3] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155 (cit. on pp. 26, 27, 34).

[4] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. *Finetuned Language Models Are Zero-Shot Learners*. 2021. arXiv: 2109.01652 [cs.CL]. URL: https://arxiv.org/abs/2109.01652 (cit. on pp. 26, 27).

[5] Song Wang, Yaochen Zhu, Haochen Liu, Zaiyi Zheng, Chen Chen, and Jundong Li. *Knowledge Editing for Large Language Models: A Survey*. 2023. arXiv: 2310.16218 [cs.CL]. URL: https://arxiv.org/abs/2310.16218 (cit. on p. 28).

[6] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. *Locating and Editing Factual Associations in GPT*. 2022. arXiv: 2202.05262 [cs.CL]. URL: https://arxiv.org/abs/2202.05262 (cit. on p. 28).

[7] Kevin Meng, Arya Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. *Mass-Editing Memory in a Transformer*. 2022. arXiv: 2210.07229 [cs.CL]. URL: https://arxiv.org/abs/2210.07229 (cit. on p. 28).

[8] Cheng-Hsun Hsueh, Paul Kuo-Ming Huang, Tzu-Han Lin, Che-Wei Liao, Hung-Chieh Fang, Chao-Wei Huang, and Yun-Nung Chen. *Editing the Mind of Giants: An In-Depth Exploration of Pitfalls of Knowledge Editing in Large Language Models*. 2024. arXiv: 2406.01436 [cs.CL]. URL: https://arxiv.org/abs/2406.01436 (cit. on p. 28).

[9] James Kirkpatrick et al. «Overcoming catastrophic forgetting in neural networks». In: *Proceedings of the National Academy of Sciences (PNAS)*. 2017 (cit. on p. 29).

[10] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. *Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?* 2022. arXiv: 2202.12837 [cs.CL]. URL: https://arxiv.org/abs/2202.12837 (cit. on pp. 29, 31, 32, 34).

[11] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. «Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing». In: *ACM Computing Surveys* (2023) (cit. on pp. 31, 34).

[12] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. *Calibrate Before Use: Improving Few-Shot Performance of Language Models*. 2021. arXiv: 2102.09690 [cs.CL]. URL: https://arxiv.org/abs/2102.09690 (cit. on p. 33).

[13] Tao Yu et al. «Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task». In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018, pp. 3911–3921. DOI: 10.18653/v1/D18-1425 (cit. on p. 37).

[14] Victor Zhong, Caiming Xiong, and Richard Socher. *Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning*. 2017. DOI: 10.48550/arXiv.1709.00103. arXiv: 1709.00103 [cs.CL]. URL: https://arxiv.org/abs/1709.00103 (cit. on p. 39).

[15] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. «RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers». In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020, pp. 7567–7578. DOI: 10.18653/v1/2020.acl-main.677 (cit. on p. 41).

[16] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2022. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903 (cit. on p. 45).

[17] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2022. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629 (cit. on pp. 45, 47, 50).

[18] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. *Toolformer: Language Models Can Teach Themselves to Use Tools*. 2023. arXiv: 2302.04761 [cs.CL]. URL: https://arxiv.org/abs/2302.04761 (cit. on pp. 45, 47).