HEXGEN-TEXT2SQL: Optimizing LLM Inference Request Scheduling for Agentic Text-to-SQL Workflows

You Peng[1*], Youhe Jiang[1*], Chen Wang[2], and Binhang Yuan[†1]

[1]Hong Kong University of Science and Technology
[2]Tsinghua University

Abstract

Recent advances in leveraging the agentic paradigm of large language models (LLMs) utilization have significantly enhanced Text-to-SQL capabilities, enabling users without specialized database expertise to query data intuitively. However, deploying these agentic LLM-based Text-to-SQL systems in production poses substantial challenges due to their inherently multi-stage workflows, stringent latency constraints, and potentially heterogeneous GPU infrastructure in enterprise environments. Current LLM serving frameworks lack effective mechanisms for handling interdependent inference tasks, dynamic latency variability, and resource heterogeneity, leading to suboptimal performance and frequent service-level objective (SLO) violations. In this paper, we introduce HEXGEN-TEXT2SQL, a novel framework designed explicitly to schedule and execute agentic multi-stage LLM-based Text-to-SQL workflows on heterogeneous GPU clusters that handle multi-tenant end-to-end queries. HEXGEN-TEXT2SQL introduce a hierarchical scheduling approach combining global workload-balanced task dispatching and local adaptive urgency-guided prioritization, guided by a systematic analysis of agentic Text-to-SQL workflows. Additionally, we propose a lightweight simulation-based method for tuning critical scheduling hyperparameters, further enhancing robustness and adaptability. Our extensive evaluation on realistic Text-to-SQL benchmarks demonstrates that HEXGEN-TEXT2SQL significantly outperforms state-of-the-art LLM serving frameworks. Specifically, HEXGEN-TEXT2SQL reduces latency deadlines by up to 1.67$\times$ (average: 1.41$\times$) and improves system throughput by up to 1.75$\times$ (average: 1.65$\times$) compared to vLLM under diverse, realistic workload conditions. Our code is available at Relaxed-System-Lab/Hexgen-Flow.

1 Introduction

Recent advances in large language models (LLMs) have led to significant breakthroughs in Text-to-SQL — the task of translating natural language questions into SQL queries [1, 2, 3, 4]. Such advancements have the potential to democratize advanced usage of relational databases by empowering non-expert users to query data intuitively, without manually crafting complex SQL statements. However, deploying agentic LLM-based Text-to-SQL systems in production environments requires more than leveraging state-of-the-art LLMs [5, 6, 7, 8, 9] with advanced agentic inference-time scaling algorithms [10, 11] — it also necessitates an *efficient inference serving infrastructure*. Specifically, such infrastructure must manage workflows consisting of multiple interdependent LLM inference requests with stringent latency and throughput demands, particularly when deployed on enterprise GPU clusters exhibiting some computational *heterogeneity*. In this paper, we address the challenge of *efficiently scheduling and executing agentic multi-stage LLM-based Text-to-SQL workloads within computational heterogeneous, multi-tenant serving environments*.

An efficient serving infrastructure for agentic LLM-based Text-to-SQL workflows is crucial for real-world deployments, particularly in enterprise production environments characterized by stringent service-level objectives (SLOs) for query response times. Meeting these SLOs is challenging due to the inherently multi-stage nature of agentic LLM-based Text-to-SQL paradigms, where each user-issued Text-to-SQL query triggers multiple interdependent LLM inference requests and subsequent database interactions. For instance, a single natural language query may generate several candidate SQL statements concurrently, each produced via distinct LLM prompts. If execution errors occur, the system iteratively invokes the LLM to refine the query, potentially requiring multiple rounds of corrections — sometimes up to ten iterations. While this multi-stage pipeline is essential to achieve high accuracy, it significantly increases computational complexity, latency sensitivity, and scheduling complexity.

Consequently, a production-level system serving numerous concurrent end-to-end Text-to-SQL queries must effectively schedule each LLM inference request from different stages onto heterogeneous GPU resources, which comprise multiple LLM model instances with potentially varying capacities for processing LLM requests. Such a scheduler must judiciously assign tasks, determining both the allocation of LLM inference requests to appropriate GPU instances and their execution order inside each model instance, to ensure adherence to per-query deadlines while maximizing overall system throughput. Addressing this scheduling challenge is vital: poor scheduling decisions can dramatically degrade response times, negatively impacting user experience and undermining the practical utility of natural-language-driven database interfaces. The scheduling problem is inherently non-trivial, as simplistic scheduling approaches suitable for less complex workloads fail to manage the dynamic dependencies, latency variability, and resource heterogeneity inherent in Text-to-SQL serving.

Optimizing LLM inference request scheduling for agentic Text-to-SQL workloads is particularly challenging due to several interrelated complexities, as we enumerated below:

- **LLM inference requests dependencies**: The state-of-the-art Text-to-SQL agents inherently involve multiple interdependent stages, each possessing distinct urgency levels. Later-stage tasks, such as the final SQL validation, cannot commence until the preceding stages are completed. Consequently, delays in early inference stages diminish the available slack for subsequent tasks, increasing the risk of end-to-end deadline violations.
- **Heterogeneity of LLM inference requests**: The LLM inference requests in the Text-to-SQL workflow exhibit substantial variability in different stages, driven by differences in the length of the query prompt and the number of output tokens generated. Such heterogeneity makes execution latencies unpredictable, complicating effective scheduling.
- **Heterogeneity of model instance serving capacity**: Enterprise production environments commonly leverage heterogeneous GPUs with different computational capabilities[1]. Consequently, the throughput and latency for an LLM inference request can vary significantly depending on the GPU specifications on which the model instance(s) execute.
- **SLO constraints in multi-tenant scenario**: Production deployments must handle continuous streams of concurrent end-to-end Text-to-SQL queries from multiple users, each with a corresponding different SLO. Thus, the scheduling strategy must be able to accommodate various priorities in a production environment.

Due to these intertwined factors, simple scheduling approaches, such as round-robin dispatching or first-come-first-served (FCFS) queues, implemented by popular LLM serving systems, would be ineffective in practice. These naive methods overlook critical task dependencies, latency variability, and GPU heterogene-

[1]In the current exciting era of generative AI, chip vendors typically release new generations of AI chips every 24 months. For example, Nvidia introduced the Turing architecture in 2018 [12], Ampere in 2020 [13], Hopper in 2022 [14], and Blackwell in Q4, 2024 [15]. On the other hand, one particular version of an AI chip often remains in use by enterprises for a much longer period.

ity, leading to suboptimal resource utilization, frequent SLO violations, and degraded system responsiveness under realistic workloads.

Notice that existing LLM serving frameworks primarily target independent LLM inference tasks, neglecting complex end-to-end workflows with multiple stages and their inherent dependencies. General-purpose schedulers typically treat requests in isolation, lacking effective coordination of dependent subtasks or enforcement of end-to-end deadlines. Recent work exploring adaptive batching [16], priority-aware request allocation [17, 18], and GPU load balancing [19] often assumes independence among LLM inference tasks and inadequately addresses heterogeneous workflows. Consequently, scenarios that require task preemption to avoid deadline violations remain unaddressed. The unique combination of multi-stage pipeline dependencies and GPU resource heterogeneity in agentic LLM-based Text-to-SQL serving is still largely unexplored, highlighting a critical gap in current serving infrastructures. To address these challenges, we introduce HEXGEN-TEXT2SQL, a novel framework designed to efficiently schedule and execute agentic Text-to-SQL workloads in heterogeneous GPU-serving environments, supporting multi-tenant queries. Our contributions can be highlighted with the following summarization:

Contribution 1. We propose HEXGEN-TEXT2SQL, a novel framework for agentic LLM-based Text-to-SQL serving, guided by a careful analysis of the workflows. The design explicitly accommodates the multi-stage nature of Text-to-SQL pipelines, managing inter-stage dependencies while exploiting parallelism across independent subtasks. By structuring HEXGEN-TEXT2SQL to support the agentic LLM reasoning loop, we enable efficient progression of sequential and parallel inference tasks, reducing idle times between dependent stages. The architecture incorporates a global coordination layer and per-instance execution management to seamlessly orchestrate the flow of tasks across heterogeneous GPU resources. This analysis-driven design establishes a robust foundation for meeting stringent SLOs in multi-tenant environments.

Contribution 2. We design a novel two-level scheduling algorithms that efficiently coordinate LLM inference requests across a pool of model replicas that comprise multiple LLM model serving instances with diverse LLM request processing capabilities. At the global level, a workload-balanced dispatcher assigns each incoming LLM inference task to the most suitable model instance, accounting for processing capabilities and current load on that model instance. The local priority queue in each model instance employs an adaptive urgency-guided queueing policy that dynamically prioritizes tasks based on their remaining deadline slack and estimated execution time. This hierarchical scheduling strategy ensures that urgent inference stages can preempt less critical tasks when necessary, allowing HEXGEN-TEXT2SQL to meet strict per-query SLOs even under heavy multi-tenant workloads. We further employ a simulator-driven approach to determine some key hyperparameters, which makes the algorithms robust across diverse workload patterns. Together, these scheduling innovations enable HEXGEN-TEXT2SQL to fully harness heterogeneous hardware parallelism while achieving consistently low latency and high throughput.

Contribution 3. We conduct a comprehensive experimental evaluation of HEXGEN-TEXT2SQL to demonstrate its performance on realistic agentic LLM-based Text-to-SQL workloads. Our experiments deploy HEXGEN-TEXT2SQL on a heterogeneous GPU cluster and compare it against state-of-the-art LLM serving systems. The results show that HEXGEN-TEXT2SQL consistently meets strict service-level objectives, significantly reducing query response times and improving throughput compared to existing solutions. Specifically, HEXGEN-TEXT2SQL reduces latency deadlines by up to 1.67× (average: 1.41×) and improves system throughput by up to 1.75× (average: 1.65×) compared to vLLM under diverse, realistic workload conditions. We also observe that HEXGEN-TEXT2SQL 's scheduling strategies ensure efficient resource utilization and robust performance in multi-tenant scenarios with diverse query complexities. Overall, the study confirms that HEXGEN-TEXT2SQL 's architecture and algorithms translate into substantial improvements in end-to-end Text-to-SQL serving performance.

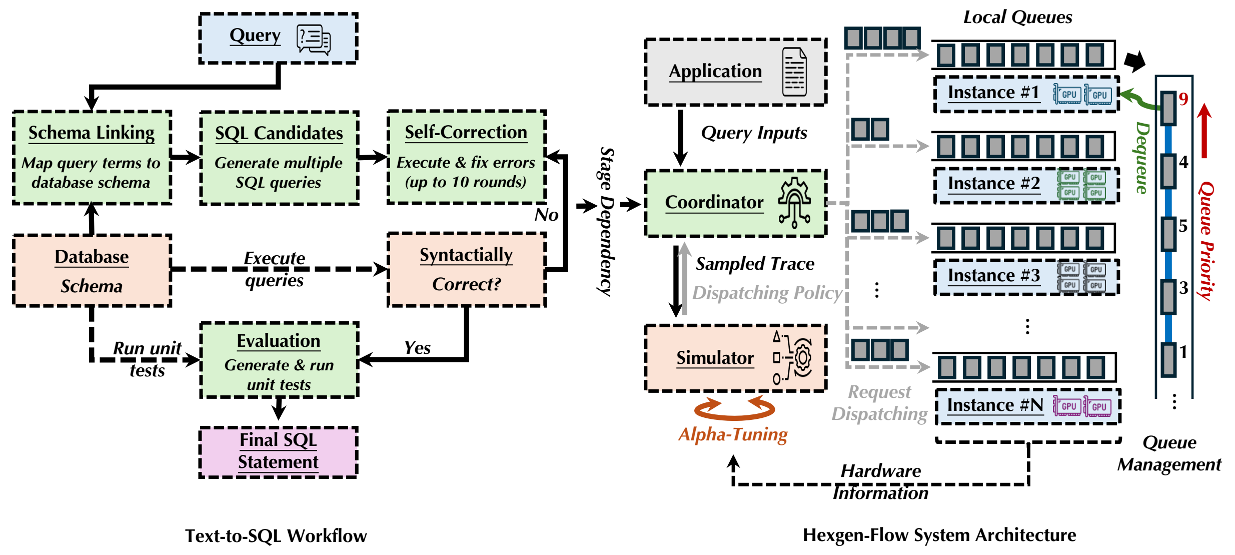

Figure 1: Text-to-SQL workflow and HEXGEN-TEXT2SQL system architecture. The Text-to-SQL workflow provides the inter-stage dependency. Incoming LLM inference requests are dispatched by a global coordinator to model instances based on workload balance and task suitability. Each model instance manages its queue using an urgency-guided priority mechanism.

2 Preliminaries

State-of-the-art Text-to-SQL serving systems face unique challenges in serving end-to-end latency-sensitive queries, requiring coordinated execution across multiple stages with interdependent LLM inference requests. In this section, we first formalize the key concepts underlying our serving system design: Section 2.1 decomposes the Text-to-SQL workflow into its constituent stages, highlighting the sequential dependencies and multi-stage pattern that necessitate specialized scheduling; Section 2.2 analyzes limitations of the current scheduling and queuing policy in existing LLM serving systems, demonstrating why general-purpose schedulers fail to meet the end-to-end latency requirements of the agentic LLM-based Text-to-SQL workflow.

2.1 Agentic LLM-based Text-to-SQL Workflow

The agentic LLM-based Text-to-SQL workflow involves several key stages to transform a natural language query into an executable SQL statement. As shown in Figure 1, we summarize the key stages in the state-of-the-art agentic Text-to-SQL paradigm (mainly following Chess [20]) as below:

- **Schema linking**: Given metadata and detailed column descriptions for each table, the LLM identifies and aligns entities mentioned in the user's natural language query to relevant tables and columns within the database schema. This step is essential to accurately ground the user's query in the underlying database structure.
- **SQL candidates generation**: Utilizing schema alignments from the previous step, the LLM generates candidate SQL queries from the natural language query. Multiple LLM inference requests are executed concurrently, employing different prompts and illustrative examples, to produce a diverse set of candidate queries. This parallel approach aims to capture multiple plausible interpretations of the user's intent.
- **Self-correction**: Candidate SQL queries are executed against the database, and any resulting execution errors prompt iterative refinement throughout subsequent LLM invocations. The system can iteratively

refine queries up to a predefined limit (e.g., 10 iterations), systematically improving their accuracy and correctness.

- **Evaluation**: After self-correction ensures syntactic correctness, the LLM generates multiple unit tests based on natural language derived from the original query. The finalized SQL candidates are evaluated against these tests, selecting the query that successfully passes the largest number of cases. These tests verify both the semantic accuracy and functional correctness of the SQL candidates, ensuring alignment with the user's original intent.

2.2 LLM Serving Queue Management

Current popular LLM serving frameworks, such as vLLM [21], Text Generation Inference (TGI) from Hugging Face [22], and TensorRT-LLM from NVIDIA [23], employ various scheduling and queueing strategies optimized for general-purpose LLM serving. For example, vLLM utilizes continuous batching [24] with a first-come-first-served (FCFS) policy, allowing new LLM inference requests to join ongoing batches during decoding. TGI groups incoming LLM inference requests based on prompt length and generation parameters to optimize batch formation. TensorRT-LLM implements an "in-flight" batching strategy, managing concurrent LLM inference requests through the scheduling policies implemented in Triton [25], including FIFO and priority-based queuing. While effective for standard LLM tasks, these frameworks are not inherently designed to handle the complexities of multi-stage, dependency-aware agentic workflows like Text-to-SQL.

Advanced queuing methods, such as Queue Management for LLM serving (QLM) [26] and the Virtual Token Counter (VTC) [27], have been proposed to address specific performance and fairness requirements — QLM introduces priority scheduling to meet Service Level Objectives (SLOs) by prioritizing urgent LLM inference requests and employing techniques like preemption and state swapping; VTC ensures equitable resource allocation among users by tracking the number of tokens served and prioritizing those with lower consumption. However, these mechanisms still can not inherently account for the nuances of the Text-to-SQL pipeline, such as varying computational costs across different stages or the impact of query latency on downstream database performance.

Concretely, we summarize the limitations that the existing queueing strategies face when applied to agentic Text-to-SQL workflows:

- **Lack of LLM inference dependency awareness**: Current queuing systems do not manage dependencies between sequential stages (e.g., schema linking before SQL generation), leading to potential inefficiencies in processing multi-stage queries.
- **Request scheduling ignores heterogeneity**: existing queueing policies treat all LLM inference requests uniformly, failing to account for the diverse resource requirements of different Text-to-SQL queries and LLM requests capacity among model instances with varying computation power, which can result in suboptimal resource utilization and difficulty meeting SLOs.
- **Inadequate SLO management in multi-tenant environments**: General-purpose LLM queues often lack the fine-grained control needed to prioritize heterogeneous LLM inference requests within the Text-to-SQL workflow, making it challenging to meet per-Text-to-SQL-query SLOs in multi-tenant settings.

To address these challenges, a novel queueing method tailored to the multi-stage and dependency-aware nature of Text-to-SQL is necessary. Such a system should optimize end-to-end performance and ensure stringent SLO adherence in multi-tenant environments.

3 HEXGEN-TEXT2SQL

In this section, we first introduce the design principle of Text-to-SQL LLM agentic workflow, then demonstrate the framework design.

3.1 Design Principle of Text-to-SQL Serving

Given the analysis of the existing limitations of the current LLM serving system in Section 2.2, we discuss the design principles of an agentic LLM-based Text-to-SQL system, each addressing critical challenges inherent in agentic Text-to-SQL workflows.

Principle 1. Explicit multi-stage dependency management. The agentic Text-to-SQL workflow decomposes each end-to-end user query into sequential and parallel stages, including schema linking, candidate generation, self-correction, and evaluation, each of which triggers single or multiple LLM inference requests. An effective orchestration of these requests in different stages is essential to ensure correctness and optimize performance. An efficient system should explicitly model inter-stage dependencies, enforcing the completion of prerequisite stages before initiating dependent ones. For parallelizable tasks, such a system should be able to dispatch subtasks concurrently and efficiently manage their completion, thereby minimizing idle time and enhancing throughput. This structured approach should be able to mitigate errors from out-of-order execution and address the complexity of orchestrating multi-stage workflows.

Principle 2. Heterogeneity-aware LLM inference request allocation. In production environments, hardware heterogeneity is commonplace — deployments often encompass a mix of GPUs with varying computational capabilities, memory capacities, and performance characteristics. This diversity could arise from incremental hardware upgrades or cost considerations. Consequently, efficient resource utilization requires a scheduling strategy that is aware of these differences between multiple LLM serving model instances. Noted that we assume that the allocation of multiple model instances could be effectively determined by existing systems [28, 29, 30], where the allocation can optimize the end-to-end SLO or throughput over a set of heterogeneous GPUs while each model instance could exhibit different serving capacities. An efficient system could address this by decoupling global task assignment from local execution management. For example, a global coordinator can evaluate the computational requirements of each task, such as memory footprint, expected execution time, and parallelism potential, and assign them to the most suitable hardware resources. This global perspective ensures optimal load balance and prevents resource contention. At the local level, each model instance can manage task prioritization and execution, adapting to real-time workload fluctuations. This scheduling ensures that both high-capacity and lower-capacity hardware are utilized effectively, maximizing overall system throughput and performance.

Principle 3. SLO guarantees. In production-level Text-to-SQL serving systems, adhering to each end-to-end SLOs is paramount to ensure consistent and predictable performance, especially in multi-tenant environments where diverse workloads coexist. As we state in Section 2.1, each stage in the agentic Text-to-SQL workflow contributes to the overall latency experienced by the user. Therefore, it's crucial to manage and schedule these stages with an awareness of their individual and collective impact on the SLOs. An ideal LLM inference request scheduling should be able to prioritize based on the remaining time budgets and estimated execution durations for each end-to-end Text-to-SQL query. Moreover, in multi-tenant scenarios, different users or applications may have varying SLO requirements. An ideal serving system should account for this by maintaining per-SLO tracking and adjusting scheduling policies to meet these differentiated objectives. This fine-grained control should be able to prevent scenarios where the performance of one tenant adversely affects others, thereby maintaining fairness and predictability across the system.

Collectively, we believe these principles enable us to effectively manage complex workflows, optimize resource utilization, and meet performance guarantees, thereby addressing key limitations in existing Text-to-SQL serving systems.

3.2 Framework Design

Following the design principles introduced earlier, HEXGEN-TEXT2SQL is a distributed system designed to efficiently serve multi-stage, LLM-based Text-to-SQL inference workloads in heterogeneous GPU clusters. Figure 1 provides an overview of the proposed system. HEXGEN-TEXT2SQL 's serving architecture is built around a centralized global coordinator and multiple GPU-backed LLM model instances, reflecting the system's three guiding principles. As a distributed Text-to-SQL serving system, HEXGEN-TEXT2SQL is explicitly designed to handle multi-stage inference workflows under heterogeneous GPU deployments while meeting strict per-query SLO deadlines. To achieve these goals, HEXGEN-TEXT2SQL intelligently dispatches and prioritizes LLM inference tasks in a *two-level* scheduling design. First, a global coordinator assigns each incoming LLM inference request to an appropriate model serving instance, accounting for the computation estimation of the LLM request and the availability (i.e., queuing status) of each model instance's variability. Second, each model instance manages its own priority queue of tasks, dynamically ordering pending inference steps by urgency.

We next describe how each principle is embodied in HEXGEN-TEXT2SQL 's design and why it is crucial for correct and efficient Text-to-SQL execution in depth.

Multi-stage dependency management. End-to-end Text-to-SQL queries execute as agentic workflows consisting of multiple dependent stages (e.g., schema linking, SQL generation, error correction, validation) that must occur in following their inherited dependence — Later stages cannot begin until earlier ones complete, and any delay in an early stage eats into the time budget of subsequent stages. To handle these strict inter-stage dependencies, HEXGEN-TEXT2SQL treats each incoming end-to-end Text-to-SQL query as a workflow of dependent LLM inference requests rather than as an isolated one. The global coordinator maintains an explicit representation of each query's pipeline status, and only dispatches an LLM inference request for execution when its predecessors have finished, ensuring the correct order of execution. For instance, once the schema linking step of a query completes, the coordinator immediately dispatches the next stage (i.e., generating SQL candidates); if a stage involves parallelizable subtasks, it dispatches all of them concurrently across available model serving instances to accelerate that stage's completion. This explicit dependency tracking guarantees correctness (each LLM step sees the proper inputs from prior steps) and prevents resource waste on tasks that would ultimately be invalidated by unmet prerequisites. The HEXGEN-TEXT2SQL monitors each end-to-end Text-to-SQL query's progress and updates scheduling parameters whenever a stage completes. In particular, the remaining end-to-end deadline for a query is propagated to its pending stages — shrinking their allowed execution windows in the local priority queue in the assigned model instances and thereby increasing their priority in the system. By *dynamically adapting* to the workflow's state in this way, HEXGEN-TEXT2SQL minimizes idle gaps between stages and ensures that downstream tasks do not miss deadlines due to upstream delays. This principled management of multi-stage dependencies ultimately improves both performance and reliability: queries complete faster by avoiding needless waiting, and the risk of cascading deadline violations is sharply reduced.

LLM inference request allocation. Production GPU clusters for LLM serving are often heterogeneous — where a standard scheduling algorithm [29, 30, 28] should be able to organize multiple model instances, which leads to optimal global SLOs or throughput while each model instance may exhibit different LLM serving capacities. On the other hand, the LLM inference requests in agentic Text-to-SQL workflows are also naturally heterogeneous in resource demand: depending on the input and output length, LLM inference requests launched from different stages can have widely varying execution times. HEXGEN-TEXT2SQL addresses this variability through a heterogeneity-aware scheduling that judiciously allocates each LLM inference request to the best-suited model instance. Instead of naive round-robin or FIFO assignment (which would ignore critical differences in hardware speed or current load and lead to SLO violation), the centralized coordinator in HEXGEN-TEXT2SQL employs a workload-balanced dispatching policy. For each incoming LLM inference request (which may correspond to a stage of some end-to-end Text-to-SQL query), the coordinator evaluates all available model instances and selects a target instance based on two factors: (i) the expected execution time of this task on that model instance given its request processing capabilities, and (ii) the current workload at that instance (e.g., queue length or utilization). Concretely, HEXGEN-TEXT2SQL 's scheduler maintains an empirical performance model for each GPU type and Text-to-SQL inference step. HEXGEN-TEXT2SQL estimates how quickly a given LLM inference request (e.g., a prompt of a certain length and prediction of the output length) would run on each candidate model instance, and it is aware of each device's backlog of work. Using this information, the coordinator computes a composite suitability score for each model instance, balancing the desire to send the task to the fastest possible model instance versus the need to avoid overloading any single model instance. The task is dispatched to the model instance with the highest score, i.e., the one offering the best trade-off between low expected latency and light current load. By dynamically routing LLM inference requests in this heterogeneity-aware manner, HEXGEN-TEXT2SQL achieves far better resource utilization and tail latency control than static or load-agnostic schemes. Heavier or latency-sensitive queries tend to run on more powerful model instances, while lighter tasks can fill in capacity on slower or busier devices, resulting in a balanced cluster workload. This global coordination not only improves overall throughput but also contributes to SLO compliance — it prevents situations where a slow model instance becomes a bottleneck or a fast model instance sits idle, thereby reducing the likelihood of queries missing their deadlines due to suboptimal placement. In summary, heterogeneity-aware dispatching allows HEXGEN-TEXT2SQL to capitalize on available hardware diversity for performance, while ensuring no end-to-end query's latency SLO is jeopardized by an inappropriate assignment.

Adaptive multi-tenant priority scheduling. To meet strict SLOs in a multi-tenant environment, HEXGEN-TEXT2SQL couples its intelligent dispatching with per-query urgency-aware scheduling. Each LLM serving model instance runs an *adaptive priority queue* that continually re-prioritizes pending LLM inference requests according to their urgency. Rather than processing LLM requests strictly in arrival order, each model instance always executes the task that is most urgent — defined in terms of the end-to-end Text-to-SQL query's deadline and remaining execution time. This design directly enforces per-end-to-end Text-to-SQL query SLO guarantees: it ensures that when the system is under load, those LLM inference requests that are closest to violating their end-to-end time budgets are serviced first, minimizing deadline misses. Concretely, when a new Text-to-SQL query enters the system, HEXGEN-TEXT2SQL assigns a target deadline to it based on its SLO. This total deadline is then apportioned across the query's multiple stages to derive an individual time budget for each LLM inference request. The budget allocation considers the average expected duration of each step so that, for example, a computationally expensive stage is given a larger share of the total time. The prioritized queue at the model instance uses this to calculate an urgency for the task, which grows as the task's waiting time increases or its deadline draws near. End-to-end Text-to-SQL queries with very little slack time and non-trivial execution length will thus have the highest urgency values. The priority queues continuously update each task's urgency in real time: while a LLM inference request waits in queue, its slack diminishes (increasing SLO pressure), and whenever a preceding step of the same query finishes, the remaining sub-deadlines for later steps are recomputed to account for lost time. Given this adaptive strategy, the most time-critical LLM inference request is always at the front of the queue when the model instances can take new LLM inference request. This urgency-driven scheduling mechanism is vital for meeting latency targets under heavy loads and unpredictable conditions in a multi-tenant scenario. As a result, HEXGEN-TEXT2SQL can guarantee a high SLO attainment rate — the vast majority of queries finish before their deadlines — while still keeping the model instance busy with as many requests as possible. The combination of global heterogeneity-aware dispatch and local urgency-aware execution allows HEXGEN-TEXT2SQL to deliver reliable performance, i.e., meeting per-end-to-end Text-to-SQL query deadlines.

By systematically embedding these principles into its design, HEXGEN-TEXT2SQL provides reliable, efficient, and scalable Text-to-SQL inference serving, directly addressing the unique demands of production-grade LLM-based workflows.

4 Scheduling Algorithm

In this section, we provide an in-depth introduction to the formulation and implementation of the global coordinator and local priority queue. We first formulate the scheduling problem as below:

Problem formulation. Formally, consider a sequence of Text-to-SQL queries $\{Q_1, Q_2, ...\}$ that arrives following some distribution $\mathbb{P}_Q$, i.e., $Q_i \sim \mathbb{P}_Q$; each query noted by Q_i includes a set of LLM inference requests represented by $\{q_{i,1}, q_{i,2}, ..., q_{i,j}, ...q_{i,n_i}\}$ along with an end-to-end SLO denoted by T_i^{SLO}. Given a set of N model instances, $\mathbf{M} = \{m_1, m_2, ...m_N\}$, where model instance m can process an LLM request $q_{i,j}$ with different processing time $t_{i,j}^m$ (including both queuing and computation time at the assigned model instance m), the goal of our scheduling problem is to find some LLM request allocation ϕ, where LLM inference request $q_{i,j}$ is executed by model instances $m_{i,j} \in \mathbf{M}$, where $\phi\left(q_{i,j}\right) = m_{i,j}$, which maximize the probability that each end-to-end Text-to-SQL query Q_i arrives from the distribution $\mathbb{P}_Q$ can be processed before its SLO:

$$\arg\max_{\phi} \mathbb{P}\left(\sum_{t_{i,j}} t_{i,j}^{\phi(q_{i,j})} \leq T_i^{\text{SLO}} \mid Q_i \sim \mathbb{P}_Q\right) \tag{1}$$

Unfortunately, determining an optimal queuing policy for such scheduling problems with unknown arrival and service time distributions is usually computationally intractable — The inherent uncertainty in job characteristics necessitates dynamic decision-making without complete information, rendering the scheduling problem NP-hard [31]. Thus, to achieve robust and efficient Text-to-SQL serving in heterogeneous GPU clusters and multi-tenant environments, we propose a heuristic-based solution, where we introduce a hierarchical, two-tiered scheduling algorithm that integrates global task dispatching with local deadline-driven prioritization. Concretely, our design includes the following components:

- At the *global coordination* level (illustrated in Section 4.1), we introduce a workload-balanced dispatcher that dynamically assigns each incoming LLM inference request to the most suitable model instance. This dispatcher jointly considers: (i) LLM request processing capability and (ii) the current workload assignment and queuing status of the model instances, effectively balancing computational loads and maximizing GPU resource utilization.
- At the *local priority queue* level at each LLM serving model instances (enumerated in Section 4.2), we leverage an advanced adaptive priority queueing method that continuously reorders tasks according to a deadline-aware urgency metric. This prioritization enables high-urgency tasks—those approaching their end-to-end deadlines—to preempt less critical tasks, significantly reducing the risk of SLO violations.
- Additionally, we incorporate a simulator-driven tuning mechanism (discussed in Section 4.3) that periodically adjusts the global dispatching coordination's hyper-parameters, balancing hardware-task alignment against workload distribution under varying runtime conditions.

Together, these design elements ensure (i) optimal exploitation of heterogeneous model instance serving capabilities, and (ii) reliable compliance with strict per-query latency objectives in dynamic, multi-tenant deployments.

4.1 Workload-Balanced Dispatching Policy

To efficiently serve Text-to-SQL workloads over heterogeneous GPU clusters, HEXGEN-TEXT2SQL adopts a workload-balanced dispatching policy that assigns each LLM inference request to the most appropriate LLM serving model instance. This policy jointly considers (i) the execution efficiency of each instance for the incoming LLM inference request and (ii) the current queued inference workloads on each model instance. Additionally, a tunable hyperparameter $\alpha \in [0, 1]$ is introduced to dynamically balance the trade-off between these two factors. We discuss how to tune α in Section 4.3.

Formulate the inference computation cost. For each incoming LLM inference request $q_{i,j}$, we first estimate its output length via a function $\hat{L}_{\text{out}}(q_{i,j})$ derived from its input length — our implementation is based on the prediction method introduced by Zheng et al. [32]. Based on the estimation, we get the predicted computational execution cost of $q_{i,j}$ on model instance m, denoted ${t_{\text{comp}}}_{i,j}^{m}$, by the following equation:

$$ {t_{\text{comp}}}_{i,j}^{m} = {t_{\text{prefil}}}^{m} \left(L(q_{i,j})\right) + {t_{\text{decode}}}^{m} \left(\hat{L}_{\text{out}}(q_{i,j})\right) \tag{2} $$

where ${t_{\text{prefil}}}^{m} \left(L(q_{i,j})\right)$ and ${t_{\text{decode}}}^{m} \left(\hat{L}_{\text{out}}(q_{i,j})\right)$ denote the estimated execution time of the prefill and decoding phase on model instance m based on number of input tokens and the estimated output tokens respectively.

Formulate the (maximal) queueing cost. The expected queuing time cost of $q_{i,j}$ on instance m, denoted by ${t_{\text{queue}}}_{i,j}^{m}$, is estimated by as the sum of the execution costs of all tasks currently in model instance m's queue Θ^{m}:

$$ {t_{\text{queue}}}_{i,j}^{m} = \sum_{q_{i',j'} \in \Theta^{m}} {t_{\text{comp}}}_{i',j'}^{m} \tag{3} $$

Note that ${t_{\text{queue}}}_{i,j}^{m}$ captures the potentially longest time task $q_{i,j}$ could wait before execution begins if it is dispatched to model instance m.

Select the serving model instance. Given the estimation of inference computation time and queuing time, an ideal instance has a low estimation for both of them. However, a linear combination of these two factors is problematic — the execution time is relatively predictable given the LLM inference query, while the queuing time can be aggressively adjusted based on the urgency we implement within each local priority queue at each model instance. Thus, we define the following non-linear combination as the heuristic score:

$$ \text{Score}\left(q_{i,j}, m\right) = (1-\alpha) \cdot \frac{\beta}{{t_{\text{queue}}}_{i,j}^{m}} - \alpha \cdot {t_{\text{comp}}}_{i,j}^{m} \tag{4} $$

For the LLM inference request $q_{i,j}$ and model instance m — $q_{i,j}$ is going to be dispatched to the instance with the highest score. Notice that there are two hyperparameters in this heuristic score (α and β); in our deployment, we fix the value of β by some trials while dynamically online tuning α during the serving. The weighting factor α determines the degree to which dispatching favors fast execution versus load balancing. When $\alpha = 1$, only execution speed is considered; when $\alpha = 0$, only queue depth matters. We empirically determine an optimal α via simulation, evaluating system-level SLO attainment across varying loads and configurations (see Section 4.3).

4.2 Local Priority Queue

After LLM inference requests are assigned to model instances by the workload-balanced dispatching policy, each instance manages its requests using a local priority queueing policy. This priority queue dynamically re-ranks LLM inference requests based on the urgency of their corresponding end-to-end Text-to-SQL query and real-time queueing conditions, enabling timely progression of multi-stage workflows. Formally, to determine the priority of each LLM inference request in the local queue, we allocate a per-request SLO budget $t_{i,j}^{\text{SLO}}$ for $q_{i,j}$ based on both execution cost and the remaining end-to-end deadline:

$$t_{i,j}^{\text{SLO}} = \left(T_i^{\text{SLO}} - \tau_{\text{elapsed}}^i\right) \cdot \frac{\bar{t}_{\text{comp}_{i,j}}}{\sum_{k=j}^{n_i} \bar{t}_{\text{comp}_{i,k}}} \tag{5}$$

where τ_{elapsed}^i is the time elapsed since the arrival of Q_i at the global coordinator, and $\bar{t}_{\text{comp}_{i,k}}$ is the execution cost of LLM inference $q_{i,j}$ averaged over all model instances, i.e.,

$$\bar{t}_{\text{comp}_{i,j}} = \frac{1}{N} \sum_{m \in \mathbf{M}} t_{\text{comp}_{i,j}}^{\ m}.$$

This proportional allocation ensures that more time is budgeted for costlier downstream LLM inference requests.

Formulate the urgency metric. Suppose a LLM inference request $q_{i,j}$ is dispatched to model instance m, we define the urgency $U_{i,j}$ of $q_{i,j}$ as the difference between its execution cost and the remaining SLO margin:

$$U_{i,j} = t_{\text{comp}_{i,j}}^{\ m} - \left(t_{i,j}^{\text{SLO}} - \tau_{i,j}\right) \tag{6}$$

where $t_{\text{comp}_{i,j}}^{\ m}$ is the estimated execution cost of the task on instance m as we defined in Equation 2, and $\tau_{i,j}$ denotes the actual queuing delay tracked by the local priority queue since $q_{i,j}$ entered the local queue. Note that higher urgency indicates a greater risk of SLO violation.

Local priority queue strategy. We implement a *dynamic priority adjustment* method, where the urgency scores are updated continuously to reflect system dynamics:

- *Queue aging*: $\tau_{i,j}$ grows over time as the LLM inference request waits in the queue.
- *Workflow progression*: When $q_{i,j-1}$ completes, τ_{elapsed}^i is updated, impacting future SLO allocations.

Thus, the model instance m always selects the LLM inference request with the highest urgency from the local queue:

$$q^* = \arg\max_{q_{i,j} \in \Theta^m} U_{i,j} \tag{7}$$

where Θ^m is the current set of queued LLM inference requests at model instance m. This adaptive strategy prioritizes requests at risk of missing their SLO and accounts for instance-specific execution characteristics.

4.3 α-Tuning Process

The hyperparameter α in the dispatching score function (Equation 4) governs the trade-off between execution cost and queuing cost. To adaptively tune α in response to real-time workload conditions, we implement a lightweight online parameter tuning process based on average query latency.

Initialization. At system startup, α is initialized to 0, prioritizing queue length minimization. During the first 100 seconds of operation, HEXGEN-TEXT2SQL uses this policy to serve incoming Text-to-SQL queries. In parallel, the system collects execution traces such as arrival times, queue delays, and stage durations for each LLM inference request. These are then used to simulate various α values on the fly, and the best-performing value α^* is selected based on average completion time of end-to-end Text-to-SQL queries.

Sliding window-based monitoring and update. After initialization, HEXGEN-TEXT2SQL assumes workload stationarity over short intervals and continues using the current α^*. The system monitors average latency in a 100-second sliding window. At the end of each sliding window, it computes the mean end-to-end latency $\bar{T}_{\text{new}}$ for all queries served in that period and compares it against the baseline latency $\bar{T}_{\text{ref}}$ from the previous window.

To determine whether latency has degraded significantly, we perform a one-sided two-sample t-test:

$$H_0 : \bar{T}_{\text{new}} = \bar{T}_{\text{ref}} \quad \text{vs.} \quad H_1 : \bar{T}_{\text{new}} > \bar{T}_{\text{ref}}$$

If the p-value falls below 0.01, the null hypothesis is rejected, indicating statistically significant latency regression. This triggers a re-tuning procedure using the most recent 100-second trace.

Simulation-guided optimization. Re-tuning is carried out using a trace-driven simulator that replays historical Text-to-SQL queries and evaluates performance under different α values. The optimal α^* is selected by minimizing the average simulated latency:

$$\alpha^* = \arg\min_{\alpha \in [0,1]} \frac{1}{N} \sum_{i=1}^{N} T_i(\alpha) \tag{8}$$

where $T_i(\alpha)$ denotes the simulated completion time of query Q_i under parameter α. The search follows a coarse-to-fine strategy: an initial sweep over $\alpha \in \{0.0, 0.2, \ldots, 1.0\}$ is refined with a finer-grained search in 0.1 increments around the best candidate.

Discussion of tuning overhead. The simulator executes entirely on the CPU and only incurs negligible overhead cost compared to the actual serving. In practice, tuned α^* values remain stable across adjacent time windows unless there are abrupt changes in workload patterns. This enables robust and low-overhead adaptation to evolving serving conditions, ensuring sustained low-latency performance.

5 Evaluation

To evaluate the design of HEXGEN-TEXT2SQL, we ask the following questions to analyze the end-to-end performance of our framework, as well as each component's contribution towards the overall efficiency improvement:

- *What is the end-to-end performance comparison between our Text-to-SQL specialized* HEXGEN-TEXT2SQL *and a general inference serving system?*
- *How effective is each component of the scheduling algorithm?*
- *What are the benefits and cost of α-tuning process?*

We state our experiment setup in Section 5.1, and address each question in Section 5.2, Section 5.3, and Section 5.4, respectively.

5.1 Experimental Setup

Runtime. Each end-to-end Text-to-SQL query was processed following a multi-agent framework named CHESS that, at the time of this study, represented the state-of-the-art in Text-to-SQL workflows [20]. We perform evaluations in the following setups:

- Hetero-1: This setup consists of two types of GPUs, A100 and A6000, each responsible for serving two model instances.
- Hetero-2: This setup consists of three types of GPUs: A100, L40, and A6000. A100 GPUs are responsible for serving two instances, while L40 and A6000 GPUs each serve one instance.

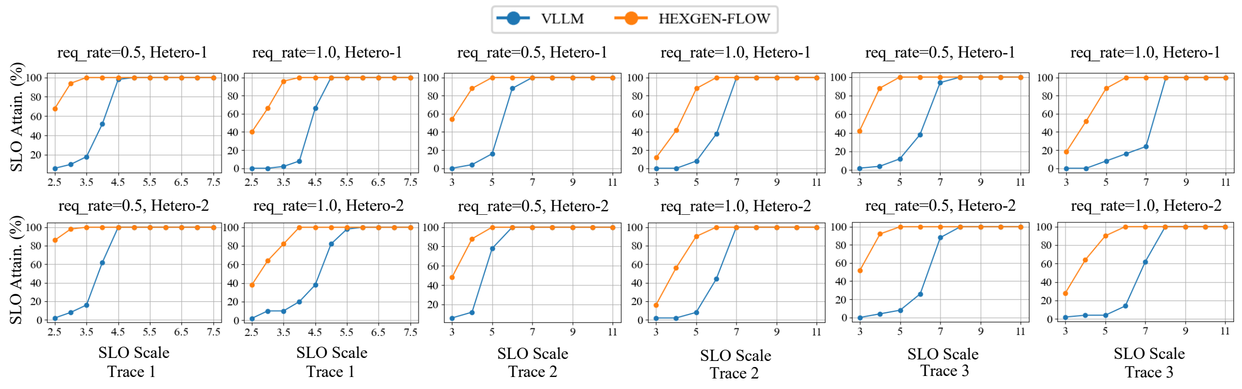

Figure 2: End-to-end SLO attainment comparison.

Note that due to the large number of model parameters, we employ a tensor parallelism degree of eight for serving all model instances using vLLM [21], and all model instances adopts continuous batching during inference.

Model and dataset. All LLM inference requests in the CHESS [20] workflow are conducted using the Llama3.1-70B model, which is a representative and popular open-source transformer model. And we follow prior work to generate three workload traces based on the development set from BIRD-bench [33], which is a cross-domain dataset designed specifically for Text-to-SQL evaluation. Our testing traces are subsampled from queries related to financial and formula1 racing databases, incorporating both simple and challenging ones. In particular, Trace 1 subsamples queries purely from the financial database, Trace 2 subsamples queries from the formula1, and Trace 3 contains queries from both databases. Depending on the complexity of the query, the workflow may require between zero and ten rounds of revision to refine the SQL query. To emulate the stochastic arrival pattern of users' Text-to-SQL queries, we send queries using a Poisson process with arrival rates of 0.5 query per second and 1.0 query per second. This modeling approach captures the inherent randomness in user interactions and aligns with methodologies employed in prior studies [18].

Baseline. To evaluate the end-to-end performance, we compare HEXGEN-TEXT2SQL with the baseline vLLM, a widely adopted inference serving system that uses First Come First Served (FCFS) to manage local queues. We dispatch the LLM inference requests based on the round-robin strategy for the baseline. This naive approach is commonly utilized in existing inference serving systems [34, 35]. For the ablation study, we compare HEXGEN-TEXT2SQL with two baseline systems: (i) RR+PQ: an intermediate design derived from our framework, implementing round-robin dispatching strategy combined with local priority queue. (ii) WB+FCFS: Another intermediate design derived from our HEXGEN-TEXT2SQL architecture, implementing workload-balanced (WB) dispatching with FCFS processing on local queues. We include these two designs as baselines to isolate and assess the impact of our workload-balanced dispatching and local priority queue independently from other enhancements in HEXGEN-TEXT2SQL.

Evaluation metrics. Following the evaluation setup of existing LLM serving frameworks [36, 37], we evaluate system performance based on SLO attainment and system throughput. The SLO is empirically determined based on the single-query processing latency, and we scale it to various multiples (SLO Scale in Figure 2) to assess performance under different levels of operational stringency. We focus on identifying the minimum SLO Scale at which the system achieves 95% and full (100%) SLO attainment.

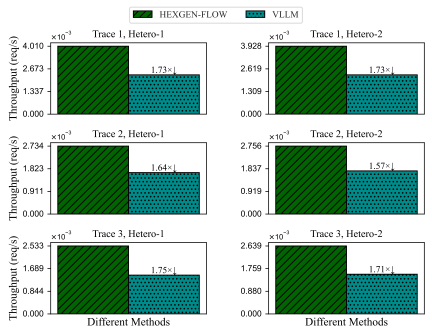

Figure 3: End-to-end system throughput comparison.

5.2 End-to-End Performance Gain

We evaluate the effectiveness of HEXGEN-TEXT2SQL by comparing its end-to-end performance with a widely adopted inference serving system, VLLM, which employs first-come-first-serve (FCFS) local queueing. The comparison focuses on two critical metrics: SLO attainment and sustained throughput, under diverse deployment traces and query arrival rates.

SLO attainment. Figure 2 presents the SLO attainment curves across multiple traces and heterogeneous configurations, under both 0.5 and 1.0 query per second workloads. Across all test conditions, HEXGEN-TEXT2SQL achieves up to $1.67\times$ and on average $1.41\times$ lower latency deadlines over 95% SLO attainment, and achieves up to $1.60\times$ and on average $1.35\times$ lower latency deadlines over 99% SLO attainment compared with vLLM. For example, in Trace 3 at 1.0 query per second, the minimum latency for HEXGEN-TEXT2SQL to achieve 95% and 99% SLO attainments are 550 and 600 seconds, whereas vLLM requires 790 and 800 seconds, which are 43% and 33% higher.

Throughput comparison. Figure 3 reports the sustained throughput achieved by both systems under a fixed 1.0 query per second arrival rate. HEXGEN-FLOW consistently delivers higher throughput across all traces and hardware configurations. The improvements range from $1.57\times$ to $1.75\times$ over VLLM, highlighting the effectiveness of our workload-balanced dispatching and urgency-aware local scheduling. For instance, in Trace 3 under Hetero-1, HEXGEN-TEXT2SQL achieves $1.75\times$ higher throughput compared to vLLM.

Summary. Together, these results demonstrate that HEXGEN-TEXT2SQL achieves significantly lower latency deadlines over 95% and 99% SLO attainments and nearly doubles throughput relative to a baseline inference serving system. The gains are attributed to the unified design of workload-balanced task dispatching and adaptive urgency-guided queueing, which allows our framework to fully exploit the parallelism and heterogeneity inherent in LLM-powered Text-to-SQL serving.

5.3 Ablation Study: Effectiveness of Scheduling

This ablation study quantifies the individual contributions of our two key scheduling innovations: the workload-balanced (WB) dispatching policy and the local priority queue (PQ). Through controlled experiments across two heterogeneous GPU deployments (Hetero-1 and Hetero-2) and varying load conditions (0.5-1.0 query per second), we demonstrate the effectiveness of both components, as well as the impact they made to the order of scheduling.

Performance of the dispatching policy. To examine the contribution from our dispatching policy, We compare the minimum latency required by WB+PQ and RR+PQ to achieve 95% and 99% SLO attainments. Figure 4 shows that replacing the round-robin routing strategy with our workload-balanced dispatching policy yields substantial improvements in serving efficiency, assuming both adopt local priority queue to manage their instances' waiting lists. Under deployments Hetero-1 and Hetero-2, and for both 0.5 query per

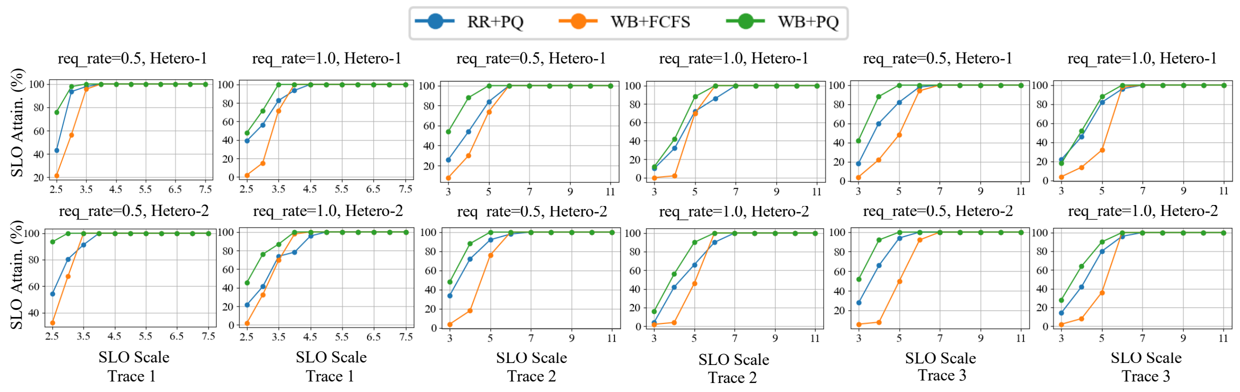

Figure 4: Ablation study of HEXGEN-TEXT2SQL's scheduling components, comparing end-to-end SLO attainment rates across: (1) round-robin dispatching + local priority queue, (2) workload-balanced dispatching + First Come First Serve, and (3) workload-balanced dispatching + local priority queue.

second and 1.0 query per second arrival rates, the WB+FCFS curves consistently shift toward lower SLO scale values compared with the RR+FCFS curve. Across all test conditions, WB+PQ achieves up to 1.38× and on average 1.18× lower latency deadlines over 95% SLO attainment, and achieves up to 1.4× and on average 1.24× lower latency deadlines over 99% SLO attainment compared with RR+PQ. For instance, in Trace 3 Hetero-1 at 0.5 query per second, all queries finish within 500 seconds under WB+PQ, compared to 700 seconds under RR+PQ. This corresponds to an approximate 40% reduction in the aspect of tail latency. These results confirm that our dispatching policy both reduces tail latency and improves GPU utilization in Text-to-SQL workloads.

Impact of the dispatching policy. For each arrived LLM inference request, our dispatching policy considers each model instance's suitability for the job, as well as the workload it carries. We control the trade-off between suitability and workload by adjusting the values of α as mentioned in §4.3, and examining the corresponding change in the SLO attainment. We demonstrate the effect of our workload-balanced dispatching policy by benchmarking task distributions before and after applying the policy for Trace 3 under the Hetero-2 configuration. As illustrated in Section 1, before applying the policy, tasks were distributed uniformly across all instances. After applying the policy, the task allocation among different instances changes accordingly. Concretely, A100 GPUs (Instances 1 and 2) handled most of the LLM inference requests from "SQL Candidates" (27.9%), "Self-Correction" (47.9%), and "Evaluation" (23.3%) stages. The A6000 GPUs (Instance 3) primarily processed tasks from "Self-Correction" stage (71.5%), while the L40 GPUs (Instance 4) focused on "Schema Linking" stage (32.3%) and "Evaluation" stage (56.3%). Consequently, this optimization resulted in a more specialized allocation of tasks among different instances, enabling the dispatching policy to route requests to appropriate model instance while balancing the load across instances, thereby improving overall system performance.

Performance of the local priority queue. To study the ablation effect from local priority queue, we let both scheduling to adopt workload-balanced dispatching, and compare the performance between WB+FCFS and WB+PQ. In figure 4, keeping Workload-Balanced scheduling unchanged, WB+PQ curves demonstrate advantages over WB+FCFS curves in all three traces, for both query arrival rates. Across all test conditions, WB+PQ achieves up to 1.5× and on average 1.2× lower latency deadlines over 95% SLO attainment, and achieves up to 1.4× and on average 1.2× lower latency deadlines over 99% SLO attainment compared with WB+FCFS. For example, under both deployment settings, when queries arrive in 0.5 query per second, the minimum latency for WB+PQ strategy in Trace 3 to achieve 95% and 99% SLO attainments are about 400 seconds and 500 seconds, whereas WB+FCFS strategy requires 600 seconds and 700 seconds, which are

Table 1: Impact of dispatching policy on task distributions. Stages 1-4 represent the aforementioned Text2SQL stages; I1–I4 represent different GPU instances, where I1 and I2 are A100 instances, I3 is an A6000 instance, and I4 is an L40 instance.

Stage	**WB dispatching**	**I1 (%)**	**I2 (%)**	**I3 (%)**	**I4 (%)**
1	before	20.8	23.1	19.9	26.8
	after	0.9	0.5	6.1	32.3
2	before	10.3	7.8	10.6	9.6
	after	27.9	29.6	16.5	3.1
3	before	26.5	23.0	25.5	21.7
	after	47.9	46.6	71.5	8.3
4	before	42.3	46.0	44.1	41.8
	after	23.3	23.3	5.8	56.3

$1.5\times$ and $1.4\times$ higher. Moreover, strategies adopt local priority queue finish 95% of queries around 20% faster than the FCFS strategies under both deployments settings when query rate is 0.5 query per second. While the extent of improvement from priority queue may vary depending on different heterogenous setting and traces, the superiority of WB+PQ suggests that dynamically prioritizing inference tasks by their remaining urgency markedly reduces head-of-line blocking and deadline misses, ensuring stable, high-quality service under varying load conditions.

Table 2: Snapshot of a local queue's state before processing next LLM call. Arrive-at represents the timestamp that the LLM call arrives.

Request ID	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Arrive-at (s)	22.4	46.3	52.3	62.4	62.8	64.4	65.0
Urgency	14.5	13.2	19.0	13.1	19.0	26.9	21.9

Impact of the local priority queue. To highlight the impact of the priority queue, we present the state of a local priority queue on one of the instances during evaluation. The snapshot, shown in Table 2, is captured immediately before the next LLM inference request to be processed by the instance. From the table, the LLM inference request $Request\#6$ arrive at 64.4, and it has the highest urgency of 26.9, so that it will be processed first by our local priority queue. In contrast, a FCFS strategy will process $Request\#1$ first because it arrives the earliest at 22.4, although it's urgency score is 14.5, which is smaller than $Request\#6$. According to the profiled execution cost, $Request\#1$ needs 26.7 seconds to finish and $Request\#6$ needs 3.2 seconds to finish. Meanwhile, the SLO allocated for $Request\#1$ is 30.4 seconds, which is larger than its profiled execution cost. On the other hand, $Request\#6$ is allocated 3.3 seconds for SLO, which is more urgent compared to $Request\#1$. This observation is consistent with our urgency score. In conclusion, the local priority queue prioritize the most urgent request first, and maximize the proportion of requests that meet their SLOs to the greatest extent possible.

5.4 Empirical Analysis of α-Tuning

Our evaluation demonstrates how dynamic α-tuning adapts to both hardware heterogeneity and workload characteristics to optimize system performance. Through controlled experiments across three distinct workload traces and two heterogeneous GPU deployments, we analyze: (1) how does a well tuned α value improve the 95% SLO attainment rate, and (2) the practical feasibility of simulation-based tuning during live serving. All experiments maintain a constant query arrival rate of 0.5 query per second while varying α

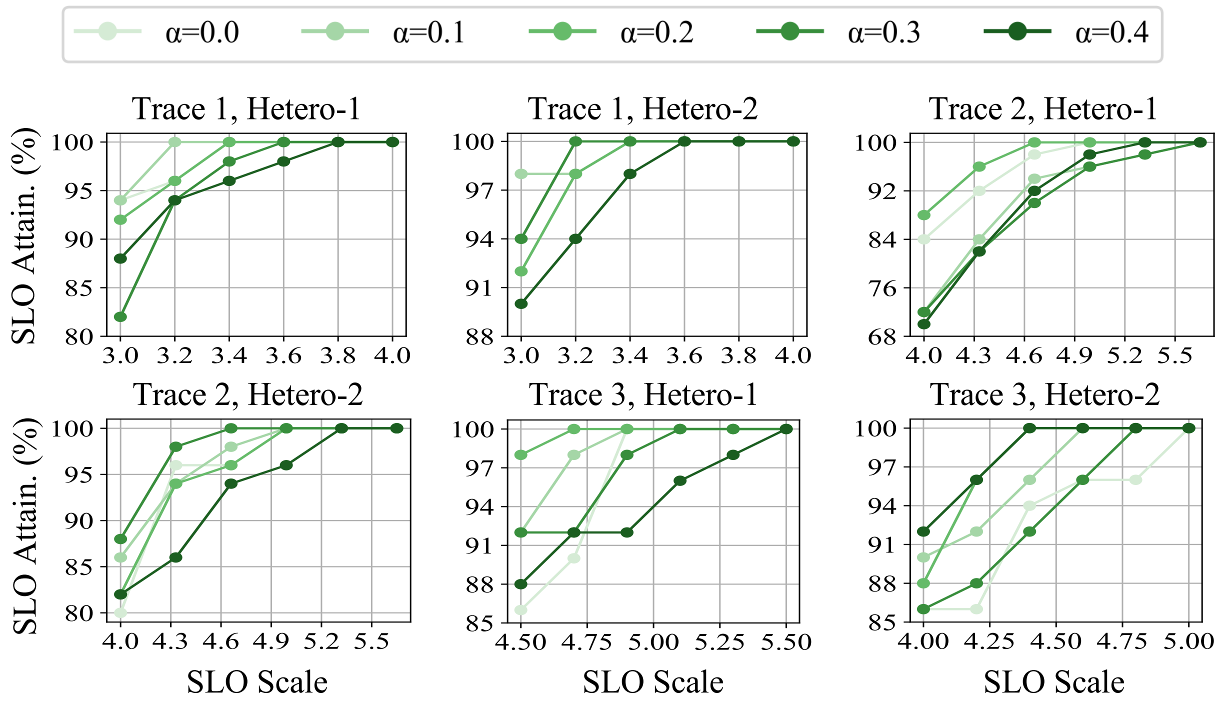

Figure 5: Performance of HEXGEN-TEXT2SQL under various α settings.

from 0 (pure workload balancing) to 0.5 (balanced weighting).

Effect of α-tuning. Figure 5 illustrates the changes in performance due to different α while keeping other hyperparameters the same. All experiments conducted on three traces assume the Text-to-SQL query arrives at 0.5 queries per second. The two experiments on Trace 1 show that taking $\alpha = 0.1$ and 0.3 gives the best results under Hetero-1 and Hetero-2 settings, respectively. Similarly, experimental results on Trace 2 indicate a better performance of $\alpha = 0.2$ under Hetero-1 deployment and $\alpha = 0.3$ in the case of Hetero-2. For Trace 3, $\alpha = 0.2$ wins over others when there are only 2 types of GPUs. And under the Hetero-2 setting, it's noticeable that when $\alpha = 0.4$, 95% of queries are finished 14% faster than the scheduling considering solely on workload balancing factor. From these three sets of results, we conclude that although the influence of task suitability doesn't outweigh the workload-balancing factor, it still can boost the performance further. Furthermore, the optimal value of α depends not only on the heterogeneous setting but also on the workload of incoming Text-to-SQL queries. Therefore, if the associated overhead is manageable, determining the optimal value of α through simulation is advantageous for enhancing system performance.

Table 3: Overhead of alpha-tuning for different experimental setups and traces.

Setup	**Trace 1**	**Trace 2**	**Trace 3**
(Hetero-2, req_rate=0.5)	124.6s	122.3s	142.2s
(Hetero-2, req_rate=1.0)	133.3s	137.8s	149.7s
(Hetero-3, req_rate=0.5)	124.7s	141.3s	158.0s
(Hetero-3, req_rate=1.0)	115.6s	133.4s	154.2s

Overhead of α-tuning. We evaluate the time cost of α-tuning simulations to assess their feasibility during real-time Text-to-SQL inference serving. Each simulation systematically tests different values of α following the α-tuning process mentioned in §4.3. As shown in Table 3, across various heterogeneous setups and workload traces, the α-tuning simulation time ranges from 115 to 158 seconds. This overhead is manageable in practice, as it is significantly shorter than the hourly timescale over which real-world workload variations typically occur [37].

6 Related Work

In this Section, we provide a brief summary about the recent advances in LLM-based Text-to-SQL approaches in Section 6.1 and then discuss some serving system design and implementation for LLM inference requests in Section 6.2.

6.1 LLM Advances for Text-to-SQL

LLMs have emerged as a revolutionary paradigm for the Text-to-SQL task [38, 1], where the core technique lies in effective SQL generation and schema linking.

SQL generation. Some pioneer studies focus on better *prompting strategies and LLM adaptations* to boost Text-to-SQL performance. Gu et al. [39] propose a structure- and content-based prompt learning method to enhance few-shot Text-to-SQL translation, while Li et al. [40] build an open-source LLM specialized for SQL generation. Other approaches fine-tune or constrain LLM outputs to improve accuracy: Ren et al. [41] propose Purple, which refines a general LLM to make it a more effective SQL writer, and Sun et al. [42] adapt a pretrained model (PaLM) specifically for Text-to-SQL tasks, achieving higher execution accuracy. In addition, Xie et al. [43] propose OpenSearch-SQL, which dynamically retrieves few-shot exemplars from a query archive and enforces consistency checks to align the LLM's output with the database schema.

Beyond simple LLM adaptations, recent *agentic approaches* leverage multiple LLM inference requests to collaboratively accomplish the Text-to-SQL tasks. Fan et al. [44] combine a small language model with a large one to handle different sub-tasks of Text-to-SQL, thereby improving zero-shot robustness. Similarly, Pourreza et al. [45] explore task decomposition across models: DTS-SQL breaks the problem into stages handled by smaller LLMs sequentially, and their DIN-SQL approach has an LLM refine its own output through iterative self-correction [46] in the prompt. Another line of research enhances the reasoning process of LLMs to produce correct SQL. One strategy is to incorporate intermediate steps or a reasoning framework during inference. Zhang et al. [47] apply the ReAct paradigm [48] to table question answering, which encourages the LLM to generate and reason with intermediate actions (e.g., decomposition or calculations) before finalizing the SQL query. Mao et al. [49] propose rewriting the user question for clarity and using execution feedback to iteratively refine the generated SQL (execution-guided refinement). To improve the chances of getting a correct query, Pourreza et al. [50] introduce a multi-path reasoning approach (Chase-SQL) that produces diverse SQL candidates and ranks them using a preference model, while Li et al. [51] propose Alpha-SQL that employs a Monte Carlo tree search to explore different query constructions in a zero-shot setting. Techniques have also been explored to optimize the context given to the LLM: Talaei et al. [20] present CHESS, which harnesses contextual information efficiently to guide the LLM's SQL synthesis without increasing model size or complexity.

Schema linking. Integrating database schema knowledge and domain-specific information into LLM-driven Text-to-SQL is another important procedure. Eyal et al. [52] decompose the natural language question and the corresponding SQL query into semantic sub-units, improving the model's understanding of how question clauses align with schema elements. Dou et al. [53] incorporate external knowledge (e.g., business rules or formulas) into the parsing process to handle queries that require facts beyond the database content. Several works specifically target schema linking challenges in the age of LLMs. Liu et al. [54] propose Solid-SQL, which uses enhanced in-context examples to make an LLM more robust at matching question terms to the schema during generation. To supply relevant schema context on the fly for open-domain Text-to-SQL, Zhang et al. [55] develop a retrieval approach that finds the pertinent tables across a large corpus and provides them to the LLM before SQL generation. Yang et al. [56] take a different approach to aid schema linking: they generate a preliminary SQL (SQL-to-schema) to identify which schema items are likely to be involved, and then use that information to guide the final query production.

6.2 LLM Inference Request Scheduling

Efficient scheduling of LLM inference requests is crucial in modern AI infrastructure, essential to meeting latency and throughput requirements, particularly under varying system constraints.

In environments with *consistent hardware and model setups*, scheduling techniques focus on optimizing latency and throughput. Patke et al. [17] introduce QLM, a system that estimates request waiting times to prioritize tasks, ensuring that SLOs are met under load conditions. Gong et al. [57] propose the future scheduler, which uses historical workload patterns and predictive modeling to make informed scheduling decisions, fulfilling SLA guarantees. Fu et al. [58] frame LLM scheduling as a learning-to-rank problem, training models to order queued requests to optimize end-to-end latency and throughput, outperforming traditional heuristics. Agrawal et al. [16] present Sarathi-Serve, a system that adjusts batching and resource allocation to balance throughput and latency, particularly effective for requests of high priority versus low priority. In setups with varying hardware capabilities and model types, recently proposed scheduling strategies adapt to resource *heterogeneity*: Wan et al. [59] develop BROS, a system that differentiates between real-time and best-effort LLM queries, ensuring interactive queries are prioritized without compromising background batch processing. Jain et al. [19] introduce a performance-aware load balancer that monitors query characteristics and system load to dynamically distribute requests across model replicas and GPU nodes. Gao et al. [60] present Apt-Serve, which employs a hybrid cache combining GPU memory and lower-tier storage to scale LLM serving while maintaining frequently-used model states in faster memory. Ao et al. [61] propose a fluid-model-guided scheduler that allocates inference tasks to approximate an ideal fluid fair share of GPU memory over time, enhancing throughput and reducing tail latency under memory pressure. Sun et al. [62] introduce Llumnix, a dynamic scheduling system that adjusts resource allocation for LLM serving in real time as query load patterns change, demonstrating benefits in single-model scenarios. While significant advancements have been made in no-stage LLM serving, multi-stage pipelines remain less explored. Very recently, Fang et al. [63] investigate the efficiency of multi-LLM application pipelines in an offline setting, using sampling and simulation to optimize inference plans for workflows involving multiple LLMs or sequential model calls.

Despite these efforts, there is a notable gap in scheduling strategies that coordinate end-to-end pipelines with multiple dependent LLM request serving stages. Our approach builds upon existing insights, such as urgency-aware prioritization and hardware-sensitive allocation, but effectively extends them to manage complex, multi-stage Text-to-SQL workflows under strict latency requirements and system heterogeneity with a significant performance boost.

7 Conclusion

In this paper, we introduced HEXGEN-TEXT2SQL, a novel inference-serving framework specifically designed to efficiently schedule and execute multi-stage agentic Text-to-SQL workloads in heterogeneous GPU clusters handling multi-tenant requests. By systematically analyzing the unique characteristics of these workloads—including stage dependencies, task variability, resource heterogeneity, and stringent latency constraints — we formulated a hierarchical scheduling solution composed of a global workload-balanced dispatcher and a local urgency-driven queue in each LLM serving model instances. This two-tiered scheduling strategy significantly improves resource utilization and effectively mitigates deadline violations. Comprehensive experiments demonstrated that HEXGEN-TEXT2SQL consistently outperforms the existing state-of-the-art LLM serving framework, achieving substantial reductions in query latency and notable improvements in system throughput under realistic production workloads — HEXGEN-TEXT2SQL reduces latency deadlines by up to 1.67× (average: 1.41×) and improves system throughput by up to 1.75× (average: 1.65×) compared to vLLM under diverse, realistic workload conditions. Overall, HEXGEN-TEXT2SQL provides an efficient infrastructure solution, enabling practical and efficient deployment of agentic LLM-based Text-to-SQL systems in real-world enterprise environments.

References

[1] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145, 2024.

[2] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. The dawn of natural language to sql: Are we fully ready? *Proceedings of the VLDB Endowment*, 17(11):3318–3331, 2024.

[3] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, et al. Dr. spider: A diagnostic evaluation benchmark towards text-to-sql robustness. In *The Eleventh International Conference on Learning Representations*.

[4] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. Catsql: Towards real world natural language to sql applications. *Proceedings of the VLDB Endowment*, 16(6):1534–1547, 2023.

[5] OpenAI. Openai gpt-4o, 2024.

[6] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[7] Anthropic. The claude 3 model family: Opus, sonnet, haiku, 2024.

[8] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[9] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[10] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

[11] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

[12] Nvidia. Nvidia reinvents computer graphics with turing architecture, 2018.

[13] Nvidia. Nvidia's new ampere data center gpu in full production, 2020.

[14] Nvidia. Nvidia announces hopper architecture, the next generation of accelerated computing, 2022.

[15] Nvidia. Nvidia blackwell platform arrives to power a new era of computing, 2024.

[16] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[17] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 18–35, 2024.

[18] Zhibin Wang, Shipeng Li, Yuhang Zhou, Xue Li, Rong Gu, Nguyen Cam-Tu, Chen Tian, and Sheng Zhong. Revisiting slo and goodput metrics in llm serving. *arXiv preprint arXiv:2410.14257*, 2024.

[19] Kunal Jain, Anjaly Parayil, Ankur Mallick, Esha Choukse, Xiaoting Qin, Jue Zhang, Íñigo Goiri, Rujia Wang, Chetan Bansal, Victor Rühle, et al. Performance aware llm load balancer for mixed workloads. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, pages 19–30, 2025.

[20] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[22] HuggingFace. Text generation inference. `https://huggingface.co/docs/text-generation-inference/index`, 2023.

[23] Nvidia. Tensorrt-llm. `https://github.com/NVIDIA/TensorRT-LLM`, 2023.

[24] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[25] Nvidia. Nvidia triton inference server, 2025.

[26] Archit Patke, Dhemath Reddy, Saurabh Jha, Haoran Qiu, Christian Pinto, Chandra Narayanaswami, Zbigniew Kalbarczyk, and Ravishankar Iyer. Queue management for slo-oriented large language model serving. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pages 18–35, 2024.

[27] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, 2024.

[28] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 586–602, 2025.

[29] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of large language model over heterogeneous environment. In *International Conference on Machine Learning*, pages 21946–21961. PMLR, 2024.

[30] Youhe Jiang, Ran Yan, and Binhang Yuan. Hexgen-2: Disaggregated generative inference of llms in heterogeneous environment. *arXiv preprint arXiv:2502.07903*, 2025.

[31] U Narayan Bhat. *An introduction to queueing theory: modeling and analysis in applications*, volume 36. Springer, 2008.

[32] Zangwei Zheng, Xiaozhe Ren, Fuzhao Xue, Yang Luo, Xin Jiang, and Yang You. Response length perception and sequence scheduling: An llm-empowered llm inference pipeline. *Advances in Neural Information Processing Systems*, 36:65517–65530, 2023.

[33] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

[34] Douglas G. Down and Rong Wu. Multi-layered round robin routing for parallel servers. *Queueing Systems*, 53(4):177–188, August 2006.

[35] Isaac Grosof, Ziv Scully, and Mor Harchol-Balter. Load balancing guardrails: Keeping your heavy traffic on the road to low response times. *arXiv preprint arXiv:1905.03439*, 2019.

[36] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[37] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[38] George Katsogiannis-Meimarakis and Georgia Koutrika. A survey on deep learning approaches for text-to-sql. *The VLDB Journal*, 32(4):905–936, 2023.

[39] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. Few-shot text-to-sql translation using structure and content prompt learning. *Proceedings of the ACM on Management of Data*, 1(2):1–28, 2023.

[40] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28, 2024.

[41] Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and X Sean Wang. Purple: Making a large language model a better sql writer. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 15–28. IEEE, 2024.

[42] Ruoxi Sun, Sercan O Arik, Alexandre Muzio, Lesly Miculicich, Satya Kesav Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, et al. Sql-palm: Improved large language model adaptation for text-to-sql. *Transactions on Machine Learning Research*.

[43] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *arXiv preprint arXiv:2502.14913*, 2025.

[44] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. Combining small language models and large language models for zero-shot nl2sql. *Proceedings of the VLDB Endowment*, 17(11):2750–2763, 2024.

[45] Mohammadreza Pourreza and Davood Rafiei. Dts-sql: Decomposed text-to-sql with small large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 8212–8220, 2024.

[46] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems*, 36:36339–36348, 2023.

[47] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M Patel. Reactable: Enhancing react for table question answering. *Proceedings of the VLDB Endowment*, 17(8):1981–1994, 2024.

[48] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

[49] Wenxin Mao, Ruiqi Wang, Jiyu Guo, Jichuan Zeng, Cuiyun Gao, Peiyi Han, and Chuanyi Liu. Enhancing text-to-sql parsing through question rewriting and execution-guided refinement. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 2009–2024, 2024.

[50] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*, 2024.

[51] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. Alpha-sql: Zero-shot text-to-sql using monte carlo tree search. *arXiv preprint arXiv:2502.17248*, 2025.

[52] Ben Eyal, Amir Bachar, Ophir Haroche, Moran Mahabi, and Michael Elhadad. Semantic decomposition of question and sql for text-to-sql parsing. In *2023 Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13629–13645. Association for Computational Linguistics (ACL), 2023.

[53] Longxu Dou, Yan Gao, Xuqi Liu, Mingyang Pan, Dingzirui Wang, Wanxiang Che, Dechen Zhan, Min-Yen Kan, and Jian-Guang Lou. Towards knowledge-intensive text-to-sql semantic parsing with formulaic knowledge. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5240–5253, 2022.

[54] Geling Liu, Yunzhi Tan, Ruichao Zhong, Yuanzhen Xie, Lingchen Zhao, Qian Wang, Bo Hu, and Zang Li. Solid-sql: Enhanced schema-linking based in-context learning for robust text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 9793–9803, 2025.

[55] Xuanliang Zhang, Dingzirui Wang, Longxu Dou, Qingfu Zhu, and Wanxiang Che. Murre: Multi-hop table retrieval with removal for open-domain text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 5789–5806, 2025.

[56] Sun Yang, Qiong Su, Zhishuai Li, Ziyue Li, Hangyu Mao, Chenxi Liu, and Rui Zhao. Sql-to-schema enhances schema linking in text-to-sql. In *International Conference on Database and Expert Systems Applications*, pages 139–145. Springer, 2024.

[57] Ruihao Gong, Shihao Bai, Siyu Wu, Yunqian Fan, Zaijun Wang, Xiuhong Li, Hailong Yang, and Xianglong Liu. Past-future scheduler for llm serving under sla guarantees. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 798–813, 2025.

[58] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient llm scheduling by learning to rank. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

[59] Wan Borui, Zhao Juntao, Jiang Chenyu, Guo Chuanxiong, and Wu Chuan. Efficient llm serving on hybrid real-time and best-effort requests. *arXiv preprint arXiv:2504.09590*, 2025.

[60] Shihong Gao, Xin Zhang, Yanyan Shen, and Lei Chen. Apt-serve: Adaptive request scheduling on hybrid cache for scalable llm inference serving. *arXiv preprint arXiv:2504.07494*, 2025.

[61] Ruicheng Ao, Gan Luo, David Simchi-Levi, and Xinshang Wang. Optimizing llm inference: Fluid-guided online scheduling with memory constraints. *arXiv preprint arXiv:2504.11320*, 2025.

[62] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 173–191, 2024.

[63] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. Improving the end-to-end efficiency of offline inference for multi-llm applications based on sampling and simulation. *arXiv preprint arXiv:2503.16893*, 2025.