Opti-Q: A Constraint-Based Optimization Framework for Multi-LLM Question Planning

Aamir Hamid[1], Bharg Barot[1], Tim Finin[1], Primal Pappachan[2], and Roberto Yus[1]

[1]University of Maryland, Baltimore County, USA. [2]Portland State University, USA.

Abstract

While large language models (LLMs) enable strong question answering (QA), budgeted deployment is complicated by nondeterminism and heterogeneous resource profiles (cost, latency, and energy). We present Opti-Q, a database-inspired, cost-based optimizer that implements a *plan-before-execute* paradigm for multi-LLM orchestration. Opti-Q models LLM invocations as physical operators in an execution DAG and, for each question, searches for plans that optimize answer quality (QoA) while trading off financial cost, latency, and energy under hard user constraints. Plans can include sequential operators that pass intermediate answers as context and parallel/blend operators that run models concurrently and merge their outputs. To search this space without executing each candidate plan, Opti-Q uses PerfDB, a statistics catalog populated and refreshed from benchmarks and execution traces, to estimate the QoA and resource costs of both individual operators and composed subplans. Using these estimates, Opti-Q performs Pareto-frontier search and selects a final plan according to user preferences. On MMLU-Pro and SimpleQA under strict budgets, Opti-Q improves average QoA by 57.69% and 41.30% over strong baselines at comparable cost, demonstrating that database-style planning yields better quality–resource trade-offs for multi-LLM QA.

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/Aamir7693/Opti-Q.git.

1 Introduction

Large Language Models (LLMs) such as GPT-5 [36], Llama 3 [18], and DeepSeek-R1 [11] have transformed natural language processing (NLP) [12], driving advances in question-answering (QA) [5], text summarization [29] and domain-specific reasoning [25]. Their success has fueled adoption across industries, from customer service to healthcare [3, 17]. However, real-world deployment remains challenging due to nondeterminism leading to hallucinations [20], high computational costs [10], substantial energy consumption [43], and latency trade-offs [38]. Considerable effort has focused on these issues, such as increasing accuracy and reliability through advanced prompting strategies (e.g., Chain-of-Thought) or extensive fine-tuning [50]. However, a fundamental challenge persists: different LLMs exhibit varying performance across question/task types [49]. Moreover, evidence suggests that relying on a single, powerful LLM for all questions is often infeasible and that a coordinated set of LLMs can outperform the single "best" model on real-world tasks [16].

Inspired by machine learning (ML) ensembles [14], recent work explores combining multiple LLMs to improve reliability and answer quality [2, 44], as cross-model validation helps mitigate hallucinations and raise response quality [13]. However, naive execution strategies (e.g., always querying all models and combining (*blending*) [24]– their outputs, or always choosing a fixed "strong" subset of models) introduce new challenges: higher financial costs and latency, inefficient resource use, and sometimes lower answer quality. Moreover, while many systems rely on dynamic sequential orchestration, where later model choices depend on earlier model outputs (e.g., via LangChain-style chains) [28] or execution-time optimization for cost/throughput [31]. Such decisions can be myopic when made without planning for downstream consequences. For example, in our experiments, a history question was best answered by a plan using Gemma-3:27B, Qwen-2.5:14B, and Phi-4:14B (outperforming other plans by a 1.24x to 1.48× factor), whereas a sports question was best handled by a simple parallel execution of Gemma-3:27B and Qwen-2.5:14B. These variations show that the optimal model combination and execution strategy are question-dependent. A robust multi-LLM QA system must therefore determine the best execution plan dynamically for each question, while accounting for end-to-end cost, latency, energy, and quality.

Database systems routinely face a similar problem: the same declarative query can be executed by many alternative physical plans, and the optimizer uses a cost model and statistics to select a plan before execution [33]. In Multi-LLM QA, many alternative LLM workflows may be possible for a user question, each trading off monetary cost, latency, energy, and answer quality. We therefore cast multi-LLM orchestration as a cost-based (and multi-objective) query planning problem, where the question serves as a query, each LLM invocation is a physical operator, and an end-to-end workflow is a physical plan whose selection is guided by a model "catalog" of historical performance statistics. However, unlike classical query optimization, multi-LLM planning must account for stochastic and semantic operator outputs (quality is uncertain and non-monotonic), output-dependent downstream costs (e.g., token lengths and content can amplify later latency/cost/energy), and inherently multi-dimensional objectives (quality–cost–latency–energy) under user constraints. This necessitates new cost estimation models and multi-objective plan selection beyond traditional single-metric cost minimization.

To tackle these challenges, we introduce Opti-Q, a cost-based optimizer for multi-LLM QA that selects an execution plan *before* issuing any model calls, mirroring how DBMS optimizers choose physical query plans prior to execution. As illustrated in Figure 1,

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

the plans generated by Opti-Q involve combinations of parallel and sequential multi-LLM operations. To select among candidate plans, we propose a technique to estimate, before execution, a plan's cost (financial, latency, energy) and benefit (answer quality) from historical performance statistics on LLMs and their combinations across question types. Using these estimates, Opti-Q identifies plans representing optimal trade-offs among competing objectives under optional user constraints via multi-objective optimization (MOO). Given the potentially large plan space, which depends on the number of LLMs considered and the complexity of the plan structure, Opti-Q employs a "pluggable optimization engine" that supports diverse planning strategies, including dynamic programming (DP), Hill Climbing, and Non-dominated Sorting Genetic Algorithm (NSGA-II) to explore candidate plans and select one for execution. This enables Opti-Q to dynamically balance performance and resource efficiency before committing to costly LLM invocations. The main contributions of this work are:

- A cost/benefit formulation of multi-LLM QA planning that selects a workflow under budget and resource constraints.
- A statistics-driven optimizer that enumerates and prunes sequential/parallel/hybrid workflows and estimates quality and resource costs prior to execution.
- A system that integrates the optimizer with an execution engine for real-time routing across multiple open source LLMs.

We evaluate our approach on two QA benchmarks, one open-ended and one multiple-choice, and compare it against four state-of-the-art baselines. In both data sets, we achieve superior QoA-cost efficiency, improving the average QoA by 57.69% and 41.30% over the best baseline at comparable cost. We further analyze Opti-Q's behavior by evaluating the impact of LLM diversity, plan complexity, and budget levels on the trade-offs it discovers.

2 Related Work

Recent work at the intersection of data management and LLMs spans two largely orthogonal directions. The first studies how LLMs can assist database query optimization (e.g., improving cardinality estimation [21], cost modeling [51], or physical operator selection [54]). While important, this line optimizes database queries over structured data and does not address our setting, where the challenge is to select and compose multiple LLM calls to answer a natural-language question under quality–cost–latency–energy trade-offs. The second direction, most related to our work, treats LLM calls as expensive and probabilistic operators in query-like workflows, and asks how to optimize these AI-powered workflows using system and optimizer principles. Orthogonal to this, orchestration frameworks such as LangChain[1] and DSPy [26] support multi-step flows but leave plan structure and physical choices largely developer-scripted, making optimization implicit.

Early approaches to LLM call orchestration adopt *cascading*, routing requests through models of increasing capacity. FrugalGPT [6] typifies this paradigm by learning a budget-aware cascade that reduces cost while maintaining accuracy. However, cascades often impose a largely fixed structure and can accumulate sequential latency; moreover, their intermediate generations can inflate downstream token usage (and thus cost/latency), making greedy "start-cheap" strategies globally suboptimal. Other ensemble methods, such as LLM-Blender [24] and LLM-TOPLA [46], improve quality by fusing multiple outputs, often assuming additional inference cost is acceptable. Building on these insights, adaptive *question-time* orchestration chooses models conditioned on the input. ThriftLLM [22] learns per-question ensemble selection within a budget, while Shekhar et al. [42] predict quality pre-inference to choose cost-effective models under latency constraints for summarization. These methods align computation with question difficulty, but they typically (i) optimize a single dominant objective and/or (ii) treat model invocations as largely independent decisions rather than as a structured plan with interactions among heterogeneous LLMs.

Complementary recent work formalizes LLM calls as operators in declarative data systems, enabling optimizer-style reasoning over AI pipelines. Galois [41] emphasizes logical optimizations (e.g., pushing down filters to reduce expensive calls), while declarative frameworks such as PALIMPZEST and Abacus [31, 40] select physical implementations based on offline calibration over representative workloads. In contrast, our focus is on question-time plan construction for multi-LLM QA: given an incoming question, we generate and evaluate alternative sequential/parallel/hybrid workflows before issuing any model calls, enabling coordinated reasoning patterns rather than selecting a single best model or a fixed pipeline.

Our work is closest in spirit to classical cost-based query optimization (i.e., enumerating alternative physical plans, estimating their behavior using statistics, and selecting plans that best satisfy resource constraints) [33]. However, multi-LLM QA departs from traditional database optimization in two key ways: (i) plans are *not* required to be semantics-preserving in the database sense (i.e., **answer quality** becomes an explicit optimization target) and (ii) operator behavior is stochastic and output-dependent (e.g., intermediate verbosity can change downstream cost). These differences make the optimization inherently multi-objective, where users may trade quality against cost, latency, and energy. Relatedly, the database community has studied multi-objective query optimization (MOQO), developing randomized and approximation techniques to efficiently compute Pareto-optimal plan frontiers, including cloud-aware settings that capture trade-offs such as runtime versus monetary cost [47, 48]. In the LLM workflow setting, a few recent LLM orchestration systems (e.g., PALIMPZEST [31]) acknowledge multi-objective trade-offs but approximate them via scalarized weighted sums. Unlike these, our approach treats cost, latency, quality, and energy as first-class objectives, explicitly exploring the Pareto-optimal frontier to honor user-defined priorities at runtime.

3 Opti-Q Multi-LLM Planning

This section formalizes user questions, LLMs, and multi-LLM question plans. Then, it postulates the selection of an optimal plan for a given question under QoA, financial, latency, and energy constraints as a multi-objective optimization (MOO) problem. Finally, it sketches our approach.

[1] https://github.com/langchain-ai/langchain

3.1 Modeling Multi-LLM Planning

Question model. We model each user question as a tuple $Q = (p_t, T, F_{\max}, L_{\max}, E_{\max}, \text{QoA}_{\min}, W)$, where pt is the textual prompt, explicitly formulated as a question (e.g., "Who are currently the oldest and youngest presidents of an EU country?"), T represents its topic(s) (e.g., "politics"). The parameters $F_{\max}$, $L_{\max}$, $E_{\max}$, and $QoA_{\min}$ specify user constraints that represent the maximum allowable total financial cost (USD/question), maximum total latency (seconds/question), maximum energy usage (J/question), and minimum acceptable QoA[2] (on a scale 0-1, 1 being the highest). We collectively refer to these as the *budget* $\mathcal{B}$. The weight vector $W = (w_F, w_L, w_E, w_{\text{QoA}})$, with $\sum_i w_i = 1$, encodes relative importance across objectives. Importantly, $\mathcal{B}$ defines *feasibility* (candidate plans must satisfy the hard constraints), whereas W is applied only after planning to select one plan from the Pareto-optimal set; W is not used to scalarize objectives during plan search. We distinguish between two question types: (1) those with *binary correctness* (e.g., multiple-choice or yes/no questions), where $QoA \in \{0, 1\}$; and (2) those with *open-ended or free-text* answers, where $QoA \in [0, 1]$ reflects graded partial correctness (e.g., "John Smith III" vs. "J. Smith"). Opti-Q's implementation provides estimators for both cases.

LLM Model. Let $L = \{L_1, L_2, \ldots, L_n\}$ be the available LLMs. Each L_i is represented by the tuple $(f_i, l_i, e_i, qoa_i, Tok_i, T_i^{\text{out}})$, where f_i, l_i, and e_i denote the expected financial cost, latency, and energy consumption of invoking L_i under an execution context; $Tok_i(\cdot)$ is the model-specific tokenizer that maps a prompt p_t to its input-token count $Tok_i(p_t)$ (e.g., GPT-4 vs. Llama-3 use different tokenizers); and T_i^{out} is the expected number of output tokens generated by L_i under the same context. We use ξ to denote the *execution context* (e.g., topic, operator type, and model). At planning time, Opti-Q instantiates the context-conditioned metrics $qoa_i(\xi)$, $f_i(\xi)$, $l_i(\xi)$, $e_i(\xi)$, and $T_i^{\text{out}}(\xi)$ using the corresponding empirical estimates retrieved from PerfDB, rather than assuming a single global quality value per model; for readability, we may omit the explicit dependence on ξ when it is clear from context. These metrics characterize the expected cost and behavior of model invocations within a plan. Each invocation cost includes a fixed and a variable component dependent on the number of input and output tokens under ξ (e.g., $f_i(\xi, p_t) = f_i^{\text{fix}}(\xi) + f_i^{\text{var}}(\xi) \cdot \left(Tok_i(p_t) + T_i^{\text{out}}(\xi)\right)$. To handle stochastic intermediate output lengths, PerfDB stores the expected output length $T_i^{\text{out}}(\xi)$ estimated from historical traces under the same context ξ, which Opti-Q uses to estimate token-dependent plan costs during planning.

Plan Model. For a question Q, let Π, denote the set of admissible plans. A plan $\pi \in \Pi$ specifies how one or more LLMs from L are invoked to process Q. Each plan comprises operations $\{\text{op}_1, \ldots, \text{op}_n\}$, where each op_i represents either an LLM invocation or a composition of LLMs, and a partial order $\prec$ defining execution dependencies. If $\text{op}_i \prec \text{op}_j$, then op_j consumes the output of op_i. Initial operations, those with no predecessors, consume Q; final operations produce the plan's answer. Operations without order constraints may execute in parallel, and all operation outputs must either feed into subsequent operations or produce final answers. For example, a plan involving operations $\text{op}_1, \text{op}_2, \text{op}_3$ may specify $\text{op}_1 \prec \text{op}_2$, where both op_1 and op_3 execute in parallel and op_2 consumes op_1's output. A *sequential operation*, denoted $\text{Seq}(Q, L_1, \ldots, L_k) = L_k\,(Q \oplus p_{ctx} \oplus L_{k-1}\,(Q \oplus p_{ctx} \oplus \cdots \oplus L_1(Q)))$, concatenates ($\oplus$) the question, a context prompt pt_{ctx} (that instructs the LLM on how to use it as a context; see Section 5), and L_1's output for input to L_2. A *parallel operation*, denoted $\text{Par}(Q, \{L_1, \ldots, L_k\}) = L_1(Q) \odot L_2(Q) \odot \cdots \odot L_k(Q)$, poses Q to each model independently and uses structured output fusion ($\odot$) to combine their outputs. $\odot$ signifies a "concatenation" in which the output of each model is prefixed with its identifier for traceability (e.g., "LLM1: answer1; LLM2: answer2"). A *blending operation* combines multiple model outputs into a single answer [24]. This operation, denoted $\text{Blend}(Q, \text{Par}(Q, \ldots), L_b) = L_b(Q \oplus pt_{bld} \oplus \text{Par}(Q, \ldots))$, uses a dedicated "blending" model, L_b, which takes the output of a parallel operation concatenated it with a blending prompt, pt_{bld} and question Q (Figure 1 shows an example; more details on the prompt in Section 5). Blending operations may appear *only* after parallel operations, and each parallel operation must be immediately followed by exactly one blending operation. Note that the same model may appear multiple times in both sequential and parallel operations, so $\text{Seq}(Q, L_1, L_1)$ and $\text{Par}(Q, \{L_2, L_2\})$ are valid.

Multi-LLM QA Planning Problem. Let π be a plan using a subset of LLMs of L to process Q. The plan induces total financial, latency, and energy consumption costs and obtains a QoA based on the chosen models and their composition. Our goal is to find Pareto-optimal plans balancing QoA, financial cost, latency, and energy under user constraints. Formally, we frame this as a MOO problem:

$$\text{Maximize} \quad \left[\text{QoA}(\pi), -\text{Financial}(\pi), -\text{Latency}(\pi), -\text{Energy}(\pi)\right]$$

$$\text{Subject to:} \quad \begin{cases} \text{Financial}(\pi) \le F_{\max} \\ \text{Latency}(\pi) \le L_{\max} \\ \text{Energy}(\pi) \le E_{\max} \\ \text{QoA}(\pi) \ge \text{QoA}_{\min} \end{cases}$$

3.2 Overview of Opti-Q Approach

Our approach generates possible question plans Π and selects an optimal one $\pi \in \Pi$ for Q that maximizes QoA while minimizing the financial/latency/energy usage costs. As shown in Figure 1, consider the example question, *"Select all that apply: Which of the following are part of the Kingdom of Denmark?"*. First, Opti-Q performs a **question parsing** phase to extract the prompt text pt, and topic(s) T (detailed in Section 5), along with user-defined constraints and priorities (budget $\mathcal{B}$ and weight vector W). We require the user to specify $\mathcal{B}$; maximizing QoA "at any cost" is supported but not our focus. If $\mathbf{W}$ is omitted, we default to uniform weights.

Bounding the plan size. Next, Opti-Q determines the maximum number of operations allowed per plan, k, based on $\mathcal{B}$. For each L_i, Opti-Q computes the maximum model invocations n_i allowed under each constraint (dividing the relevant budget by model's per-constraint cost) and taking the minimum across constraints. For example, the budget in Figure 1 constrains a resource-heavy LLM, L_1, to one call while allowing a lighter L_2 up to three calls. Limits are not mutually exclusive: a plan may call L_1 once and still use L_2

[2]We use the generic term *QoA* to denote any metric employed in prior work to assess model performance depending on the task and dataset, such as accuracy, BLEU, F1, or embedding-based similarity measures (e.g., cosine similarity).

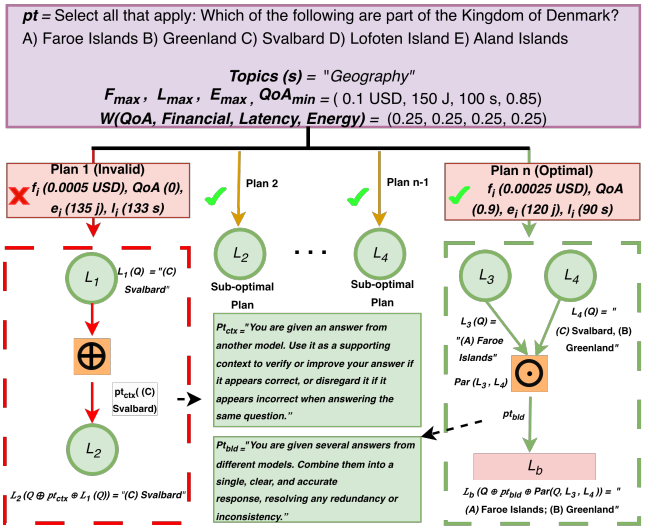


Figure 1: Comparison of candidate multi-LLM execution plans for a question under unconstrained optimization.

more or one times within the remaining budget. Opti-Q samples k uniformly from $[min_i(n_i), max_i(n_i)]$; as shown in our evaluation and in traditional ML ensemble size selection methods [15], this randomized choice within the interval does not harm performance.

Performance Statistics (PerfDB). Opti-Q includes a statistical catalog, PerfDB, that maintains performance statistics for individual LLMs and composite plans. PerfDB enables the planner to estimate plan metrics (i.e., latency, financial cost, QoA, and energy) without executing the plan. PerfDB is organized into levels of compositional depth (i.e., operator-nesting depth in the plan DAG), which determines the granularity of stored statistics. Level 0 contains statistics for single LLMs obtained from public benchmarks such as ML.Energy [9] and Artificial Analysis.[3] Level 1 aggregates results of single operations with "depth 1", e.g., $\text{Seq}(Q, L_1, L_2)$ or $\text{Par}(Q, \{L_1, L_2, L_3\})$, while Level 2 and higher store measurements for subsets of plans with "depth 2" or higher (e.g., $\text{Blend}(Q, \text{Seq}(Q, L_1, L_2), \text{Par}(Q, \{L_3, L_4\}), L_b)$). All entries are stored under a common schema keyed by execution context ξ (e.g., topic, operator and model). PerfDB is populated both offline and after plan execution. For an executed plan, Opti-Q records *observable* metrics such as financial cost and latency, obtained directly from model API responses. Energy and QoA are not typically measurable at runtime: energy data is not always exposed by APIs, and QoA requires ground truth. Accordingly, PerfDB stores *observed* values when available, and otherwise stores *predicted* values derived from proxy variables such as token counts and benchmark priors. Offline, PerfDB is populated with Level 0 entries from existing benchmark data to mitigate cold-start limitations; after execution, new observations are appended to increase coverage at higher levels. Since models and serving stacks evolve over time, Opti-Q plans using a current snapshot of PerfDB, which can be periodically refreshed with new benchmark runs and execution traces. Section 6 analyzes how varying PerfDB coverage affects planning performance.

[3]https://artificialanalysis.ai/

Plan Generation. In this phase, Opti-Q generates plans with up to k operations. Because the plan space grows exponentially in $|L|$ and k (as we will detail in the following section), an exhaustive search is infeasible. Opti-Q therefore supports a "pluggable optimization engine" capable of supporting diverse planning strategies (Section 4): (i) Dynamic Programming (DP), an exact solver that can recover the Pareto frontier for small |L| and k but suffers from state-space explosion; (ii) stochastic Hill Climbing (HC), a lightweight greedy heuristic with low planning latency that can stagnate in local optima in our non-convex multi-objective landscape; and (iii) NSGA-II, a multi-objective evolutionary algorithm that balances exploration and exploitation. This design enables both exact optimization (when feasible) and lightweight heuristics, while using NSGA-II as the default for scalable Pareto-front approximation.

Plan Execution Phase. To execute the selected plan, operations are carried out according to the plan topology, applying context propagation and blending where applicable. For example, consider the execution of *Plan n* in Figure 1, which includes a parallel call followed by blending, i.e., $\text{Blend}(Q, \text{Par}(Q, L_3, L_4), L_b)$, and is selected in this case. Models L_3 and L_4 run in parallel to answer the multi-answer user prompt. Suppose L_3 returns a partial but correct subset (e.g., (A) Faroe Islands), while L_4 returns (B) Greenland but also includes a plausible yet incorrect extra option (e.g., (C) Svalbard). The blender L_b aggregates the candidate sets, cross-checks them against the question constraints, removes unsupported options, and produces a consolidated final prediction (A, B) (Faroe Islands; Greenland). This simple example illustrates the strength of multi-objective planning: a single-model plan (only L_4) may be cheaper but less robust, risking error propagation. We quantify variability due to nondeterminism by repeated executions and confidence intervals in Section 6.

4 Plan Generation

To answer a question using multiple LLMs, we must (i) represent candidate execution plans in a compact form that supports efficient manipulation, (ii) estimate a plan's cost–benefit *without* executing it, and (iii) search the resulting plan space to find high-quality tradeoffs and select a plan that matches user preferences.

4.1 Plan Encoding

All planners in Opti-Q operate over the same compact, yet expressive encoding candidate plan π. We represent π as a tuple $(\textit{connectivity\_map}, \mathbf{m})$, where *connectivity_map* canonically encodes the plan's directed acyclic execution topology and $\mathbf{m}$ assigns one model to each of the k nodes. Figure 2 shows an example plan with its encoding. The DAG topology over k operations is encoded by the upper-triangular adjacency matrix $M \in \{0,1\}^{k \times k}$, with $M_{ij} = 1$ for $(1 \le i < j \le k)$ iff there is a data-flow edge from L_i to L_j. For computational efficiency, we flatten M into a $k(k-1)/2$-bit vector $B = (b_{12}, b_{13}, \ldots, b_{(k-1)k})$, where each bit $b_{ij} \in \{0,1\}$ indicates whether L_i feeds its output to L_j. Blending operations are implicit: a designated blending model (L_b) is invoked to combine inputs if a node L_j has an in-degree $\deg^-(L_j) > 1$. To ensure invariance under node relabeling (structural isomorphism), we enumerate all valid topological orderings of the DAG and select the lexicographically

maximal bit vector, $B_{\max}$, defining the canonical integer identifier as *connectivity_map* = integer($B_{\max}$). This encoding method guarantees acyclicity by construction, facilitates efficient manipulation, and provides a unique, isomorphism-aware representation of complex execution flows. We enforce the constraint that the final node must have an in-degree ≥ 1, ensuring the plan produces a valid output. Figure 2(A) illustrates an execution plan

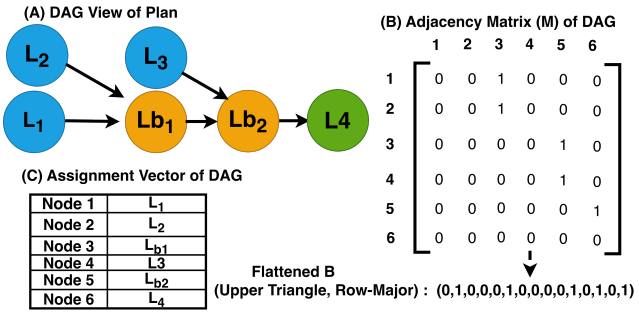


Figure 2: Sample question plan encoding.

composed of $k = 5$ operations constructed from Sequential, Parallel, and Blending operations, while Figure 2(B) and (C) show its connectivity matrix and model assignments, respectively. The plan begins with a parallel operation where models L_1 and L_2 process the question Q, defined as $op_1 = \text{Par}(Q, (L_1, L_2))$. Their outputs are then blended using model L_{b_1}, forming $op_2 = \text{Blend}(Q, op_1, L_{b_1})$. Concurrently, an independent parallel operation executes model L_3, defined as $op_3 = \text{Par}(Q, (L_3))$. The results from the first blend (op_2) and the execution of L_3 (op_3) are synthesized in a hierarchical blending step, $op_4 = \text{Blend}(Q, op_2, op_3, L_{b_2})$. Finally, in a sequential operation, $op_5 = \text{Seq}(Q, op_4, L_4)$, model L_4 processes the original question Q again using the output of op_4 as context to produce the final answer. This encoding captures the execution structure compactly and canonically, requiring $O(k^2)$ space.

4.2 Cost-Benefit Estimation

For each candidate plan, we estimate (pre-execution) its expected QoA, financial cost, energy, and latency.

Token Estimation. Cost-benefit metrics depend on the number of tokens processed and generated, which we must estimate pre-execution. For each model L, let $Tok_L(\cdot)$ denote its tokenizer (models may segment the same string differently) and let T_L^{out} be the model's expected output length (e.g., average response tokens from PerfDB). We estimate two quantities for a plan π: total generated tokens T_π^{out} and total processed tokens T_π^{proc} (input + output across model invocations).

Sequential. In $\text{Seq}(Q, L_1, L_2)$, L_1 consumes $Tok_{L_1}(Q)$ tokens and produces $T_{L_1}^{\text{out}}$. The second model consumes the original question plus a context prompt pt_{ctx} and the *immediate predecessor's* output, i.e., $Tok_{L_2}\big(Q + pt_{\text{ctx}} + T_{L_1}^{\text{out}}\big)$, and produces $T_{L_2}^{\text{out}}$. We conservatively sum outputs, $T_{\text{Seq}}^{\text{out}} = T_{L_1}^{\text{out}} + T_{L_2}^{\text{out}}$, allowing for potential redundancy where later models restate prior text. For a length-k chain $\text{Seq}(Q, L_1, \ldots, L_k)$ we avoid exponential growth by appending only the immediate predecessor's output at each step: $T_\pi^{\text{proc}} = \big(Tok_{L_1}(Q) + T_{L_1}^{\text{out}}\big) + \sum_{i=2}^{k}\Big(Tok_{L_i}\big(Q + pt_{\text{ctx}} + T_{L_{i-1}}^{\text{out}}\big) + T_{L_i}^{\text{out}}\Big)$.

Equivalently, chains can be viewed as nested compositions, e.g., $\text{Seq}(Q, L_1, L_2, L_3) = \text{Seq}(Q, \text{Seq}(Q, L_1, L_2), L_3)$, with pt_{ctx} applied from the second step onward.

Parallel + blending. In $\text{Blend}\big(Q, \text{Par}(Q, (L_1, \ldots, L_k)), L_b\big)$, each L_i processes $Tok_{L_i}(Q)$ input tokens and produces $T_{L_i}^{\text{out}}$ output tokens. The blending model L_b consumes the question, a blending prompt pt_{bld}, and the concatenated outputs: $Tok_{L_b}\Big(Q + pt_{\text{bld}} + \sum_{i=1}^{k} T_{L_i}^{\text{out}}\Big)$ and produces $T_{L_b}^{\text{out}}$. Thus, $T_\pi^{\text{proc}} = \sum_{i=1}^{k}\big(Tok_{L_i}(Q) + T_{L_i}^{\text{out}}\big) + Tok_{L_b}\Big(Q + pt_{\text{bld}} + \sum_{i=1}^{k} T_{L_i}^{\text{out}}\Big) + T_{L_b}^{\text{out}}$, $\quad T_\pi^{\text{out}} = T_{L_b}^{\text{out}}$.

For a parallel operation followed by blending, each model in the parallel set processes the same prompt concurrently. Specifically, in $\text{Blend}\big(Q, \text{Par}(Q, (L_1, \ldots, L_k)), L_b\big)$, each L_i processes $Tok_{L_i}(Q)$ input tokens and produces $T_{L_i}^{\text{out}}$ output tokens. The blending model L_b then combines these results using a blending prompt pt_{bld}, receiving $Tok_{L_b}\Big(Q + pt_{\text{bld}} + \sum_{i=1}^{k} T_{L_i}^{\text{out}}\Big)$ input tokens and producing $T_{L_b}^{\text{out}}$ output tokens. Hence, the total number of processed tokens in a parallel+blending operation is $\sum_{i=1}^{k}\big(Tok_{L_i}(Q) + T_{L_i}^{\text{out}}\big) + Tok_{L_b}\Big(Q + pt_{\text{bld}} + \sum_{i=1}^{k} T_{L_i}^{\text{out}}\Big) + T_{L_b}^{\text{out}}$, producing $T_{\text{Par}}^{\text{out}} = T_{L_b}^{\text{out}}$ output tokens.

General plans. For an arbitrary (nested) plan π, we compute $(T_\pi^{\text{proc}}, T_\pi^{\text{out}})$ by recursively applying the above rules bottom-up over the plan tree/DAG and summing the processed tokens over all invoked models. This yields a conservative, tokenizer-aware estimate of token volume that propagates through both sequential and parallel compositions.

QoA Estimation. Predicting a plan's QoA before execution is difficult due to model nondeterminism and the combinatorial space of multi-LLM plans. We therefore estimate QoA using PerfDB, a performance database populated from offline benchmarks and execution traces that stores topic-conditioned QoA statistics for (i) atomic invocations (e.g., (Q, L_i)) and (ii) common local compositions (e.g., $Seq(Q, L_i, L_j)$ and $Blend(Q, Par(Q, \{L_1, \ldots, L_k\}), L_b)$). The goal is not perfect calibration, but a conservative estimate that preserves *relative* differences between candidate plans, which is sufficient for Pareto-based search.

Topic conditioning and lookup. Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be the inferred topics of Q. For any operator or substructure op, we aggregate topic-specific entries as $QoA(op) = \frac{1}{m}\sum_{i=1}^{m} QoA_{op,T_i}$. To initialize estimation, we first attempt an exact PerfDB match for the whole plan (or large substructures). If not available, we perform a fuzzy match using a weighted similarity that combines structural overlap with a normalized edit-distance over a canonicalized plan string; we accept a fuzzy match only if the score exceeds a fixed threshold (e.g., 0.75 in our experiments). Importantly, all QoA values used during planning are retrieved from PerfDB under a single schema (topic + operator/substructure key), even when the retrieval itself is fuzzy. The complete processed is in the extended version of the paper [].

Sequential refinement. For a sequential dependency where op_j consumes the output of op_i (e.g., a local $\text{Seq}(\cdot, L_i, L_j)$), we update the running estimate using a pairwise relative-effect factor derived from PerfDB: $QoA_{\text{next}} = \text{clip}_{[0,1]}\Big(QoA_{\text{curr}} \cdot \frac{QoA(Q,\text{Seq}(L_i,L_j))}{\max(\epsilon, QoA(Q,L_j))}\Big)$. where ϵ is a small constant and $\text{clip}_{[0,1]}(\cdot)$ bounds the result to

$[0, 1]$. Intuitively, the ratio compares the composed step against the successor alone, capturing the marginal impact of inserting L_i before L_j under the same topic-conditioned context. *Example:* if $QoA(Q, L_j) = 0.50$ and $QoA(Q, \text{Seq}(L_i, L_j)) = 0.60$, the factor is 1.2; with $QoA_{\text{curr}} = 0.55$, we obtain $QoA_{\text{next}} \approx \text{clip}_{[0,1]}(0.55 \times 1.2) = 0.66$.

Parallel + blending. For a blend node op_{L_b}, where the blending operation is defined as $\text{Blend}(Q, \text{Par}(Q, \{L_1, \ldots, L_k\}), L_b)$, we combine the incoming branch estimates using PerfDB references for (i) the blend operator itself and (ii) each predecessor run standalone. Let $QoA^{\text{ref}}_{op_{L_b}} = QoA(Q, op_{L_b})$ and $QoA^{\text{ref}}_{L_s} = QoA(Q, L_s)$. Given the branch estimates computed in the current traversal, $QoA^{\text{new}}_{L_s}$, we scale the reference blend quality by the average relative change of its inputs:

$$QoA^{\text{new}}_{op_{L_b}} = \text{clip}_{[0,1]}\left(QoA^{\text{ref}}_{op_{L_b}} \cdot \left(1 + \frac{1}{k}\sum_{s=1}^{k} \frac{QoA^{\text{new}}_{L_s} - QoA^{\text{ref}}_{L_s}}{\max(\epsilon,\ QoA^{\text{ref}}_{L_s})}\right)\right).$$

This estimator is robust to a single weak branch (average rather than product) while still reflecting systematic improvements/degradations across branches. *Example:* suppose $k = 2$, $QoA^{\text{ref}}_{op_{L_b}} = 0.60$, and references $QoA^{\text{ref}}_{L_1} = 0.50$, $QoA^{\text{ref}}_{L_2} = 0.40$. If the current plan yields $QoA^{\text{new}}_{L_1} = 0.60$ and $QoA^{\text{new}}_{L_2} = 0.44$, then the average relative change is $\frac{1}{2}\left(\frac{0.60-0.50}{0.50} + \frac{0.44-0.40}{0.40}\right) = 0.15$, so $QoA^{\text{new}}_{op_{L_b}} = \text{clip}(0.60 \times (1 + 0.15)) = 0.69$.

Plan-level propagation and fallbacks. We compute the plan QoA by traversing the plan DAG in a topological order that respects precedence constraints (the complete algorithm is in the extended version []). So, an operator is processed only after all prerequisites are estimated. Along edges corresponding to sequential consumption we apply the sequential update rule; at blend nodes we wait for all incoming branches and apply the blending rule. When PerfDB lacks a required composed entry (e.g., missing $QoA(Q, \text{Seq}(L_i, L_j))$), we fall back to the closest available statistics (e.g., use other known predecessors of L_j to estimate an average relative-effect factor, or default to a neutral factor of 1 when no pairwise information exists). For nested blend structures where direct references are sparse, we normalize the structure by flattening nested blends before performing lookups, then apply the same blend rule on the resulting fan-in. These fallbacks ensure estimation completes under incomplete coverage; in Section 6 we quantify how estimation error varies with PerfDB coverage.

Financial Cost Estimation. We estimate the financial cost of a plan as the sum of its operation-level costs [6]. For a single invocation of model L_i, we use a simple fixed-plus-variable pricing model: a fixed per-call charge f^{fix}_i and a per-token rate f^{var}_i applied to the total number of billed tokens (prompt + generated). Using our token estimator, the expected billed tokens for op_i equal the model-specific tokenized prompt length plus the predicted output length: $Financial(Q, L_i) = f^{\text{fix}}_i + f^{\text{var}}_i \cdot \left(Tok_i(p_t) + T^{\text{out}}_{op_i}\right)$. For a parallel operator $\text{Par}(Q, \{L_1, \ldots, L_n\})$, invocations are independent, so costs add: $Financial(\text{Par}(Q, \{L_1, \ldots, L_n\})) = \sum_{i=1}^{n} Financial(Q, L_i)$. For a general plan π, we account for prompt growth in sequential and nested structures by charging each operation for its own prompt plus the outputs of its immediate predecessors: $Financial(\pi) = \sum_{i=1}^{n}\left(f^{\text{var}}_i\left(Tok_i(p_t) + \sum_{op_j \in \text{pred}(op_i)} T^{\text{out}}_{op_j} + T^{\text{out}}_{op_i}\right) + f^{\text{fix}}_i\right)$. Here, $\text{pred}(op_i)$ denotes the set of operators whose outputs are included in op_i's prompt (e.g., the immediate predecessor in a sequence, or all fan-in branches for a blend node).

Energy Consumption Estimation. LLM energy consumption is approximately proportional to the number of tokens processed and generated [53]. Let e_i (J/token) denote the energy per token for operation op_i obtained from PerfDB or external measurements [9]). We estimate plan energy by reusing the same structural accounting as financial cost, but without a fixed per-call term. This keeps energy estimation consistent with how prompts and intermediate outputs propagate through the plan.

Latency Estimation. Plan latency depends on both the plan structure and the number of tokens processed and generated [42]. For a single invocation op_i, we approximate latency as linear in token volume using a per-token time coefficient l_i (s/token) $Latency(Q, L_i) = l_i(Tok_i(p_t) + T^{\text{out}}_{op_i})$. For sequential (or more generally, precedence-constrained) portions of a plan, latencies add along executed operations, using the same prompt-growth accounting as above. For a pure parallel operator $\text{Par}(Q, \{L_1, \ldots, L_n\})$, branches execute concurrently, so the end-to-end latency is dominated by the slowest branch. For mixed plans with parallelism and joins, we apply these rules along the plan DAG: sum along each critical path and take the maximum over paths that run concurrently.

4.3 Pareto Search over Multi-LLM Plans

The plan space is inherently combinatorial. Under our operator-DAG representation, each node selects one of $|L|$ models and each of the $\binom{k}{2}$ forward edges may be present or absent, yielding $|\Pi| = |L|^k \cdot 2^{\binom{k}{2}} = |L|^k \cdot 2^{\frac{k(k-1)}{2}}$, with the derivation in the extended version []. This space becomes large even for modest settings (e.g., k=5, $|L|$=10 gives $10^5 \cdot 2^{10} \approx 1.024 \times 10^8$ plans). Moreover, optimizing over Π is NP-hard even for simplified variants of our objective structure [42]. Consequently, exhaustive enumeration is infeasible except for small instances, and we instead rely on search procedures that can efficiently explore Π using inexpensive, pre-execution objective estimates. In particular, we consider three widely used approaches: a genetic algorithm (NSGA-II), bottom-up dynamic programming (DP), and randomized hill climbing. All three optimizers operate over the same plan encoding (canonical DAG topology + model assignments), use the same PerfDB-based estimators for objective evaluation, and output a set of non-dominated feasible plans that trade off QoA, cost, latency, and energy under user budgets without enumerating all of Π. We then select a single plan for execution by normalizing objectives and applying the user weights W. Due to space constraints, we present only the core mechanics below; full plan-generation details and algorithmic variants appear in the extended version [].

Genetic Algorithm (NSGA-II). NSGA-II is a robust default for our discrete, non-convex search space (topology + model choices). Each individual encodes (i) a canonicalized DAG topology (`connectivity_map`) and (ii) a model-assignment vector $\mathbf{m}$ (one LLM per node). Starting from an initial population of size N, NSGA-II iterates for G generations using binary tournament selection (non-dominated rank,

then crowding distance). With crossover probability C_r, we apply one-point crossover independently to the topology and assignment segments, then apply targeted mutations that mirror common plan edits: (i) edge flips (rewiring), (ii) node insertions (adding an operator), and (iii) model reassignments (changing L_i on a non-blending node). After any structural edit we apply lightweight repair to enforce validity (e.g., a valid sink/output) and the blending constraint (any node with in-degree > 1 is assigned the blending model), followed by canonicalization so isomorphic plans share a single encoding. We rank the merged parent+offspring pool with fast non-dominated sorting and constraint handling (feasible dominates infeasible; infeasible ranked by total violation), then select the next generation by taking successive fronts with crowding-distance tie-breaking. The first non-dominated front $\mathcal{F}_1$ approximates the Pareto set; we choose a single execution plan by applying W to normalized objectives over $\mathcal{F}_1$.

Dynamic Programming. DP provides a systematic baseline that enumerates canonical plan structures and propagates sets of feasible assignments per structure. For plans with k operations, the DP state is indexed by the canonical structure identifier `connectivity_map`. Each bucket stores multiple candidate assignments $\mathbf{m}$ (one model per node) together with plan-level estimated costs computed via PerfDB-based estimators. Blending is enforced by construction: nodes with in-degree > 1 must use the designated blending model, while nodes with in-degree ≤ 1 may use any base model. To build $DP[k]$ from $DP[k-1]$, we extend each canonical $(k-1)$-node structure by adding a new node and enumerating predecessor subsets $S \subseteq \{0, \ldots, k-2\}$, adding edges $(u \to k-1)$ for all $u \in S$. For each parent assignment, we enumerate admissible model choices for the new node, evaluate objectives, and then canonicalize to merge isomorphic structures into the same bucket.

To control state growth, we use pruning. In particular, we found a simple *parent-specific* Δ*-QoA progress gate* to provide the best runtime–quality trade-off: when expanding a parent into children, we retain a child only if $QoA(\pi_{\text{child}}) \geq QoA(\pi_{\text{parent}}) + \Delta$. We use $\Delta = 0.05$ by default. (We also explored within-bucket approximate dominance pruning via multiplicative ε-dominance; details are in the extended version.)

Hill Climbing. DP can become state-intensive as k grows, so we also implement a randomized local-search optimizer that quickly finds high-quality tradeoffs under a per-question time budget, following [48]. The method performs repeated random restarts: each restart samples a feasible plan π, then greedily hill-climbs using *Pareto-improving* moves until reaching a locally non-dominated solution; collecting these local optima across restarts yields an approximate frontier. We define the neighborhood of π via exhaustive single-step mutations: (i) *model mutation* changes the base model at one non-blending node, (ii) *edge flip* toggles one edge in the upper-triangular encoding and keeps the result only if it remains a valid DAG, and (iii) *node addition* appends one new node (up to the operation limit k) using an admissible extension that preserves a valid sink/output. After any structural mutation we repair assignments to satisfy the blending constraint, canonicalize the topology, and remap $\mathbf{m}$ under the canonical permutation so isomorphic DAGs share a single encoding. We discard infeasible neighbors that violate hard budgets. A neighbor is accepted if it dominates the current plan (no worse on all objectives and strictly better on at least one); we scan the neighborhood and take the first dominating move, repeating until no neighbor dominates the current plan. As candidates are discovered we maintain a global frontier (optionally applying the same Δ-QoA filter from before when inserting plans).

5 Opti-Q Implementation

We implement Opti-Q as a modular framework for flexible multi-LLM QA planning under user-defined constraints[4]. Our prototype integrates five widely used open-source LLMs: Gemma (27B)[45], LLaMA3-ChatQA (8B)[32], Qwen2.5 (14B)[39], Phi-4 (14B)[1], and Mistral (7B)[23]. We selected these models for their complementary strengths across benchmarks [49, 52] (e.g., strong reasoning vs. conversational QA vs. efficiency). All models run locally via Ollama[5], which also simplifies adding or replacing models. For tuning system parameters, we construct a held-out configuration set by sampling 100 SimpleQA and 100 MMLU-Pro questions; this set is strictly disjoint from the test set in Section 6 to prevent leakage.

Blending Model. For blending we use GenFuser [24], which fuses independently generated responses into a single coherent answer while preserving diversity. GenFuser consumes concatenated inputs ($p_{bld} \oplus \text{Par}(Q, \ldots)$) and is implemented with a fine-tuned T5-XL (3B) sequence-to-sequence model from the Flan-T5 family [8], chosen for its efficiency on multi-source inputs.

Question Processing and PerfDB Initialization. Opti-Q initializes PerfDB with per-model, per-topic parameters required for planning, including model size and empirical estimates of cost, latency, energy, and QoA. We populate these parameters from public benchmarking sources (including ML.ENERGY Leaderboard [9] and Artificial Analysis[6]), and use them as priors that are refined as additional traces are collected.

At question time, Opti-Q extracts topics for an incoming question Q using BERTopic [19]. Concretely, we embed Q with a pre-trained transformer encoder, reduce dimensionality with UMAP [35], cluster with HDBSCAN [34], and derive representative topic keywords using c-TF-IDF [19]. The inferred topics are then added to the question tuple used by the planners and PerfDB lookups.

Prompt Design. We consider four prompting strategies: *Zero-Shot (ZS) [27]*, *Few-Shot (FS) [4]*, and *Chain-of-Thought (CoT) [50]*, and a *baseline* strategy (B) (i.e., just instructions without examples or reasoning). We also define two auxiliary prompts: a *context prompt* pt_{ctx} for sequential operations (ensuring the successor treats the predecessor output as evidence) and a *blending prompt* pt_{bld} for parallel operations (instructing the blender to reconcile redundant or conflicting answers).

Our prompting evaluation shows that *ZS* provides the best trade-off between efficiency and stability (see the extended version []). We therefore use *ZS* in the experiments of Section 6.

QoA Computation. We measure QoA using semantic similarity rather than lexical overlap (e.g., BLEU [37], ROUGE [30]), since LLM outputs often vary stylistically while preserving meaning.

[4] Code, datasets, prompts, and results are available at https://github.com/Aamir7693/Opti-Q.git.
[5] https://ollama.com
[6] https://artificialanalysis.ai/

For free-form QA, we embed generated and reference answers with the 384-dimensional *all-MiniLM-L6-v2* sentence transformer[7], which has demonstrated strong performance on the Massive Text Embedding Benchmark (MTEB)[8] and is widely adopted QoA is then computed as the cosine similarity between embeddings. For multiple-choice QA, we score single-answer questions (e.g., MMLU-Pro) by normalized exact match (binary in $\{0, 1\}$). For multi-answer ("select all that apply") questions, we normalize the predicted and gold option sets and compute a set-overlap score in $[0, 1]$ (e.g., Jaccard similarity) so partially correct predictions receive partial credit.

NSGA-II Hyperparameter Optimization. We tuned NSGA-II hyperparameters via grid search on a held-out stratified set (200 questions); we report the best setting and release configs/scripts. The best-performing configuration by mean scalarized score in the converged population was $P = 200$, $G = 200$, and $\boldsymbol{\mu} = \{0.3, 0.1, 0.3\}$, which consistently outperformed other settings by 1.2×–12× across validation metrics adopting this configuration for all subsequent experiments.

6 Experimental Evaluation

Evaluation setup. Our evaluation targets two foundational QA paradigms using specialized benchmarks: SimpleQA, an open-ended factual QA dataset containing 4K questions, and MMLU-Pro a domain-specific dataset comprising 12K multiple choice questions. Because exhaustive evaluation would entail (≈3.2M plan executions across five models), we instead build a representative, computationally tractable testbed using stratified sampling: we select 10 diverse question types from each dataset (e.g., *History*, *Mathematics*, *Sports*) and uniformly sample 10 questions per type, yielding 200 questions total (100 per benchmark) spanning factual, analytical, and reasoning-intensive tasks. Experiments run on a dedicated high-performance cluster equipped with 2×24-core Intel Xeon Gold 6240R (2.40 GHz, 165 W TDP) CPUs and x4 NVIDIA RTX 6000 GPUs, with shared central storage exceeding 3 PB. We report micro-averaged QoA, latency, energy, and financial cost, over five runs with confidence intervals 95%.

Planner Comparison. We compare Opti-Q's planner backends—NSGA-II, exact DP, pruned DP, Hill Climbing, and pruned Hill Climbing—as the maximum number of operations k increases. The full comparison table is provided in the extended version []. For each $k \in \{1, \ldots, 5\}$, we measure (i) planning time and (ii) Pareto-front quality using hypervolume (HV) and inverted generational distance (IGD), computed relative to the exact DP frontier (when tractable). For fair comparison under matched computational budgets, the hill-climbing planner is run with a per-question timeout equal to the maximum wall-clock time observed for NSGA-II on the same question under the same operation limit. Exact DP recovers the reference frontier but exhibits state-space explosion as k grows (from near-zero planning time at small k to over 200 s at k=5). Pruned DP substantially reduces planning time (down to seconds at k=5) at the cost of lower frontier quality. Hill Climbing, under the same time budget as NSGA-II, degrades more sharply at larger k due to local optima; adding pruning further reduces frontier quality with little runtime benefit under timeout-bounded search. NSGA-II provides the best scalability–quality trade-off, maintaining near-reference frontier quality while keeping planning time in the low tens of seconds at k=5 (21 s), motivating it as the default backend for larger plan spaces. Unless noted otherwise, in subsequent experiments, we use pruned DP as the default DP-based planner and Hill Climbing with strict Pareto dominance (i.e., without the Δ-QoA filter) as the default Hill Climbing variant, with per-question timeout matched to NSGA-II for the same question and operation limit, based on the above runtime–quality trade-offs.

Baselines and scope. Opti-Q is designed for broad open-domain QA, while most existing cost-aware multi-LLM methods target classification tasks. To enable a comparison on MMLU-Pro, we treat each question as a classification and compare Opti-Q against ThriftLLM [22], LLM-Ensemble [7], FrugalGPT [6] and LLM-Blender [24]. Because LLM-Ensemble lacks a budget constraint,we implement a budgeted variant that greedily selects top-K weighted models until the budget is met [7]. For LLM-Blender, we adopt the latest *PairRanker* and *GenFuser* components with the author-recommended settings. On SimpleQA, we compare Opti-Q to FrugalGPT and LLM-Blender using the released configurations. Unless noted, all methods (Opti-Q and baselines) were evaluated under identical budget constraints. We omit *LLM-TopLa* (repository errors), Shekhar et al. (summarization task), and *Palimpzest* (offline distillation focus) as they are not directly comparable under our QA-oriented orchestration setting.

6.1 Baseline Comparison Setup

We evaluate all baseline systems under a unified cost–accuracy framework. Each method is configured to adhere to the same strict per-question budget as Opti-Q. However, *LLM-Blender* is not budget-aware and invokes its full model set for every query, leading to different average costs across questions: ≈ 4.4×10^{-5} USD/query for the *MMLU-Pro* classification benchmark and ≈ 1.6×10^{-5} USD/query for the *SimpleQA* open-ended benchmark. Hence, it is shown in our figures for completeness but excluded from cost–accuracy comparisons, and compared against Opti-Q solely on the basis of QoA. Furthermore, we adapt FrugalGPT to operate under our strict per-query budget constraint rather than its native expected-cost scheduling mechanism to align its behavior with other baselines. All baseline systems were re-implemented using our selected model suite and benchmark datasets; we denote these adapted versions with an asterisk (e.g., *FrugalGPT**, *ThriftLLM**). Hereafter, all baseline names refer to these re-implemented variants. The evaluation spans five budget levels, $b \in \{1, \ldots, 5\}$, each scaling a baseline single-model usage cost of 1.2×10^{-5}. This value represents the average per-question execution cost across all candidate LLMs. Thus, the maximum allowable financial cost per question is defined as $F_{\max}(b) = b \cdot 1.2 \times 10^{-5}$.

Classification-reasoniong QA (MMLU-PRO). We first evaluate Opti-Q on the *MMLU-Pro* benchmark, a multiple-choice QA task where each question has fixed answer options, and performance is measured by accuracy under varying budget constraints. At the lowest budget (b=1), as shown in Figure 3(a), *ThriftLLM** is competitive with the Opti-Q planner variants (about 41%), due to its routing mixture between Mistral and Qwen models and its

[7] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[8] https://huggingface.co/spaces/mteb/leaderboard

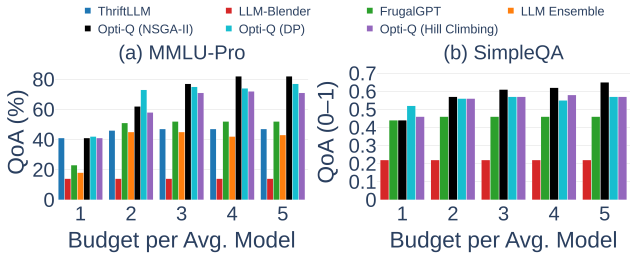


Figure 3: QoA versus budget across two benchmarks: (a) *MMLU-Pro*, where QoA is measured as the percentage of questions answered correctly, and (b) *SimpleQA*, where QoA is measured as the average quality per question.

per-question ensemble aggregation, which balance cost and model capability. Opti-Q (DP) is slightly higher at approximately 42%. As the budget increases (b=2–3), all baseline systems converge around 45–47% accuracy and subsequently plateau. This stagnation arises because these methods execute all models in parallel and deterministically collapse their predictions into a single label via confidence ranking or majority voting, thereby discarding contextual and complementary evidence among models. Consequently, additional model invocations increase the cost without improving the accuracy. In contrast, Opti-Q improves substantially with budget. At b=2, Opti-Q (DP) performs best (≈ 73%), while at b=3, Opti-Q (NSGA-II) overtakes (≈ 77% versus ≈ 75% for DP and ≈ 71% for Hill Climbing). Unlike deterministic aggregation strategies, Opti-Q constructs question-specific sequential/parallel/hybrid execution plans that preserve inter-model reasoning context and allocate budget adaptively across dependent and independent operations. This enables continued gains beyond the baseline plateau. Cost analysis shows that ThriftLLM* and FrugalGPT* consume, on average, 58–76% across budget levels, while LLM-Ensemble* consumes 53–99%. In contrast, Opti-Q uses 64–88% of the available budget through selective plan allocation while achieving substantially higher accuracy. Overall, these results demonstrate that deterministic parallel ensembles exhibit early efficiency but limited scalability, whereas adaptive planning in Opti-Q breaks this plateau by dynamically composing sequential, parallel, and hybrid plans to achieve superior cost–accuracy efficiency on classification-style QA tasks. Compared to LLM-Blender, Opti-Q attains substantially higher QoA (0.82 vs. 0.14), underscoring the benefit of adaptive execution planning.

Open-Ended QA (SimpleQA). We next evaluate Opti-Q on the *SimpleQA* benchmark (Figure 3(b)), an open-ended QA task that tests the model's ability to synthesize and articulate factual knowledge. Performance is measured by QoA, range [0,1], computed as the cosine similarity between predicted and reference embeddings. Among the baselines, FrugalGPT* remains nearly flat at around 0.46 QoA across budgets, consuming on average 58–76% of the budget (mean cost 7.2×10^{-6}–1.4×10^{-5}). LLM-Blender* (shown for completeness but excluded from cost-normalized comparisons due to fixed full-ensemble invocation) incurs a nearly fixed cost of ≈ 1.6×10^{-5} USD/query and remains substantially lower at about 0.22 QoA. In contrast, all three Opti-Q planner variants outperform these baselines. At the lowest budget (b=1), Opti-Q (DP) performs best (≈ 0.52), followed by Opti-Q (Hill Climbing) (≈ 0.46) and Opti-Q (NSGA-II) (≈ 0.44). At moderate budgets (b=2–3), the three planners converge to similar QoA (≈ 0.56–0.61). At higher budgets (b=4–5), Opti-Q (NSGA-II) achieves the best QoA, improving to about 0.65 while utilizing 64–88% of the available budget, whereas DP and Hill Climbing plateau near 0.55–0.58.

6.2 Impact of Number of Operations

We study how the maximum number of operations per plan (k) affects QoA and resource usage. For each question, we vary k from 1 to 5 and aggregate results across the five PerfDB coverage levels. This experiment serves as a sensitivity analysis on plan-space size: increasing k expands the set of candidate sequential/parallel/hybrid workflows available to the planner. Due to space constraints, the main paper focuses on the NSGA-II planner results for MMLU-Pro and SimpleQA (Figure 4 and Table 1). The extended version [] includes the figures for NSGA-II (SimpleQA) and corresponding k-sweep analyses for the DP and Hill Climbing baselines, all of which exhibit the same fundamental QoA–resource trade-offs as k scales. As expected, expanding the plan space up to k = 5 yields

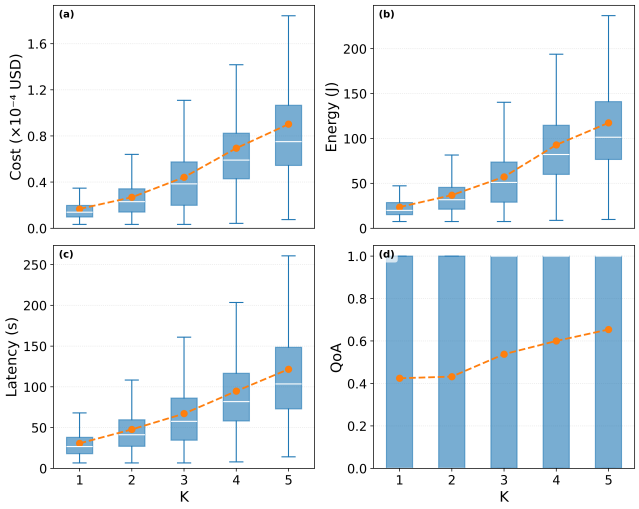


Figure 4: NSGA-II performance across operation limits k on *MMLU-Pro*. Panels show per-question distributions of (a) financial cost, (b) energy, (c) latency, and (d) QoA. Boxplots indicate the median and interquartile range; the dashed line with markers shows the mean. Increasing k improves QoA but also increases cost, energy, and latency.

the highest overall QoA, though at a predictable cost to latency, energy, and financial expense across both benchmarks. However, the nature of these QoA gains differs fundamentally depending on the benchmark's evaluation structure (Table ??). For *MMLU-Pro*, which relies on exact-match multiple-choice accuracy, the QoA distribution is effectively binary (the 0.5–0.8 bucket remains empty). Here, increasing k from 1 to 5 systematically shifts outcomes from the low-QoA bucket (57.55% → 34.58%) to the high-QoA bucket (42.45% → 65.42%). This demonstrates that larger plans effectively correct previous misclassifications, leading to strong, consistent improvements. In contrast, *SimpleQA* presents a more nuanced,

non-monotonic trend due to its open-ended nature. After an initial QoA degradation at $k = 2$ (where the low-QoA bucket spikes to 86.36%), performance steadily improves. By $k = 5$, the low-QoA bucket shrinks to 57.23%, while the middle and high-QoA buckets expand to 27.61% and 12.86%, respectively. This indicates that while larger plans consistently enhance semantic answer quality, these gains are more gradual and highly sensitive to plan composition compared to rigid exact-match tasks.

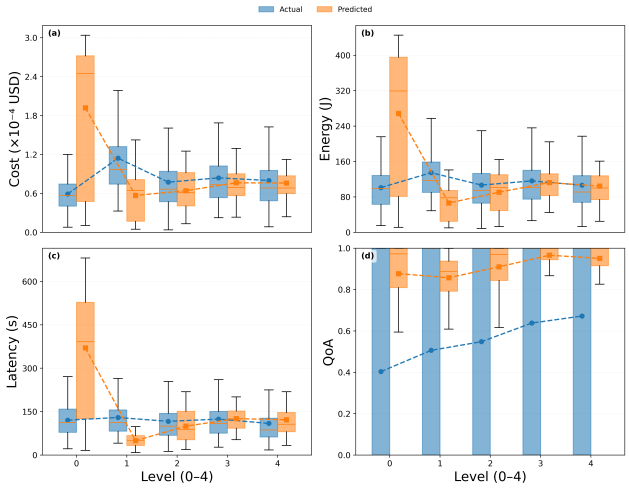
Evaluating the raw trade-offs highlights the diverging cost-benefit profiles between the two benchmarks. For SimpleQA, scaling from $k = 1$ to $k = 5$ exhibits moderate returns: a 23.6% relative QoA gain drives cost, energy, and latency up by roughly 6.5×, 6×, and 3.8×, respectively. Conversely, MMLU-Pro demonstrates a much stronger return on resource investment at higher k. For this dataset, a substantial 54.7% QoA improvement is achieved alongside 4× to 5.3× increases across the resource metrics, justifying the expanded plan space. Crucially, as k grows, the planner increasingly favors concurrent execution. Hybrid and parallel plans account for ≈ 36%, 77%, and 82% of all executions at $k = 3, 4$, and 5, respectively (leaving sequential plans at just ≈ 14% by $k = 5$). This architectural shift explains the steep rise in resource consumption at higher k. Therefore, while small-to-moderate ensembles ($k = 3$) capture the majority of the benefits efficiently for semantic tasks, maximizing accuracy on rigid classification tasks justifies the higher operational limit, actively trading cost, latency and energy for significant QoA upside.

6.3 Impact of Historical Data Complexity Levels

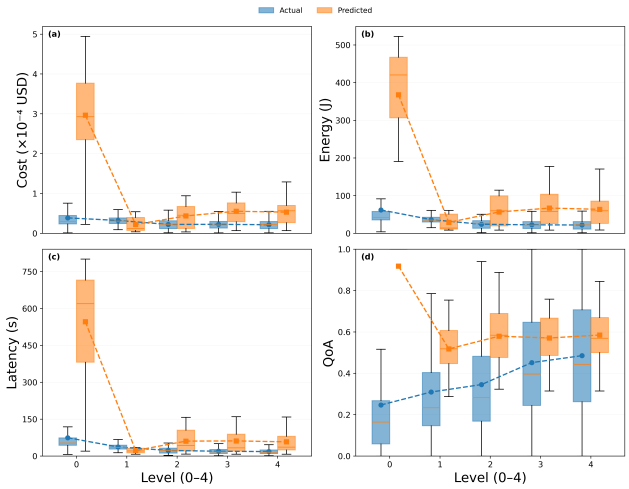
We assess how Opti-Q adapts to varying amounts of historical performance metadata in PerfDB. We define five coverage levels (0–4) that simulate progressively richer historical data availability per question type. As noted in Section 3, Level 0 contains only static model priors (i.e., a cold-start setting with no execution history), while Levels 1–4 progressively enrich PerfDB with empirical traces collected from *disjoint training questions* that share the same topical distribution as the evaluation set. This setup enables the planner to generalize from related but unseen instances while avoiding data leakage. The cumulative numbers of recorded executions available in PerfDB at Levels 1–4 are 135, 985, 1,735, and 2,360, respectively. These counts denote the total execution traces used to populate PerfDB (not evaluation questions). For each question, the planner can compose up to five operations (k=5), as identified in the previous experiment.

Due to space constraints, the main paper reports only the NSGA-II results for *MMLU-Pro* and *SimpleQA* (Figure 5a and 5b). The extended version [] includes the corresponding level-sweep analysis for DP and Hill Climbing, which show the same qualitative trend as PerfDB coverage increases. Figures 5a and 5b compare predicted versus actual plan metrics across coverage levels for *MMLU-Pro* and *SimpleQA*, respectively. Table 1 further summarizes how the distribution of plan QoA shifts across levels for each benchmark.

For *MMLU-Pro*, QoA improves monotonically from 0.40 at Level 0 to 0.67 at Level 4 (an absolute gain of +0.27, i.e., ≈ 66.7% relative improvement). Resource usage is comparatively stable after an early increase at Level 1: mean cost rises from 5.9×10^{-5} (Level 0) to 1.14×10^{-4} (Level 1), then settles in the 7.7×10^{-5}–8.4×10^{-5} range



(a) *MMLU-Pro*



(b) *SimpleQA*

Figure 5: Actual vs. predicted plan-level cost, energy, latency, and QoA across PerfDB coverage levels (0–4) for NSGA-II on MMLU-Pro (a) and SimpleQA (b). In both, calibration improves with historical coverage: early overestimation under sparse data (Level 0) diminishes at higher levels, while predictions preserve the level-wise trends needed for planner ranking decisions.

(Levels 2–4). Energy and latency show the same pattern, peaking at Level 1 (134.8 J, 129.7 s) and then remaining broadly stable (roughly 106–116 J and 109–124 s). This indicates that richer history primarily improves plan selection quality on MMLU-Pro without requiring sustained additional resource expenditure beyond early calibration. For *SimpleQA*, QoA also increases monotonically, from 0.25 at Level 0 to 0.48 at Level 4 (an absolute gain of +0.24, i.e., ≈ 96.7% relative improvement). In contrast to MMLU-Pro, resource usage *decreases* substantially as coverage improves: mean cost drops from 3.9×10^{-5} to 2.2×10^{-5}, energy from 62.1 J to 22.1 J, and latency from 74.4 s to 18.1 s (Levels 0 to 4). This suggests that empirical history enables the planner to select not only better but also more efficient

Table 1: Plan QoA distribution (%) across QoA buckets as a function of operation limit K and PerfDB coverage level on MMLU-Pro and SimpleQA. Larger K and higher levels generally shift mass toward higher-QoA buckets.

Param	Val	MMLU-Pro QoA bucket (%)			SimpleQA QoA bucket (%)		
		0–0.5	0.5–0.8	0.8–1.0	0–0.5	0.5–0.8	0.8–1.0
Operation limit K							
K	1	57.55	0.00	42.45	71.63	20.00	5.12
K	2	56.79	0.00	43.21	86.36	9.63	1.87
K	3	46.22	0.00	53.78	71.74	20.22	6.01
K	4	40.00	0.00	60.00	65.22	24.51	8.48
K	5	34.58	0.00	65.42	57.23	27.61	12.86
PerfDB coverage level							
Level	0	59.67	0.00	40.33	87.86	0.43	11.71
Level	1	49.40	0.00	50.60	80.40	13.60	6.00
Level	2	45.20	0.00	54.80	76.20	17.40	6.40
Level	3	36.20	0.00	63.80	59.00	27.40	13.60
Level	4	32.80	0.00	67.20	56.00	28.00	16.00

plans for open-ended QA, where the initial cold-start priors appear especially conservative.

Table 1 shows that richer PerfDB coverage systematically shifts plans toward higher-quality QoA buckets in both benchmarks. For *MMLU-Pro*, the high-QoA bucket (0.8–1.0) grows from 40.33% (Level 0) to 67.20% (Level 4), while the low-QoA bucket (0–0.5) drops from 59.67% to 32.80%. The middle bucket remains 0.00% at all levels, which is expected because MMLU-Pro is evaluated using exact-match multiple-choice accuracy and thus produces effectively binary QoA outcomes. For *SimpleQA*, the shift is more gradual and non-binary. The low-QoA bucket decreases from 87.86% (Level 0) to 56.00% (Level 4), while the middle bucket expands sharply from 0.43% to 28.00%, and the high-QoA bucket increases from 11.71% to 16.00%. This pattern is consistent with the open-ended nature of the task: richer history improves semantic answer quality progressively, often moving plans from poor to moderate quality before reaching the highest QoA band. Figures 5a and 5b also show that estimation quality improves with PerfDB coverage. In both benchmarks, Level 0 exhibits strong overestimation—especially for cost, energy, and latency—because the planner relies on static priors without empirical calibration. Once historical traces become available (Levels 1+), predicted resource metrics move substantially closer to the observed distributions and preserve the correct level-wise trends. QoA estimates remain more optimistic than actual QoA across levels, particularly in sparse-data settings, but the gap narrows with increased coverage. Importantly, the estimator preserves the relative ordering of plan quality across levels, which is the key requirement for effective multi-objective search. This is especially important in our setting, where the planner must estimate the joint behavior of multi-step, multi-model workflows (rather than a single LLM), under variable prompts and execution paths. In general, richer historical metadata sharply improves QoA while stabilizing resource usage, enabling the planner to generalize from related but unseen questions. These results demonstrate the value of empirical traces in guiding multi-objective scheduling under varying data availability.

6.4 Impact of Model Diversity

We evaluate the effect of *LLM diversity* (i.e., the number of distinct models invoked in a plan) on QoA and resource usage. Again, based on the results in Section 6.2, we set the maximum number of operations K=5, each calling any model from the pool L). Since NSGA-II provides the best scalability–quality trade-off among the planners in our earlier comparisons, we use NSGA-II as the default planner for this experiment. We define a plan's diversity level, d, as the count of unique models invoked across all operations within it, which can range from 1 (one LLM used) to $\min(k, |L|)$ (where k is the number of operations and $|L|$ the LLM pool's size). We report micro-averaged metrics across diversity levels using *Level-4* PerfDB coverage aggregated across all budget settings, and analyze marginal efficiency ratios ΔQoA/ΔFinancial, ΔQoA/ΔEnergy and ΔQoA/ΔLatency to quantify trade-offs (see Table 2 and Figure 6).

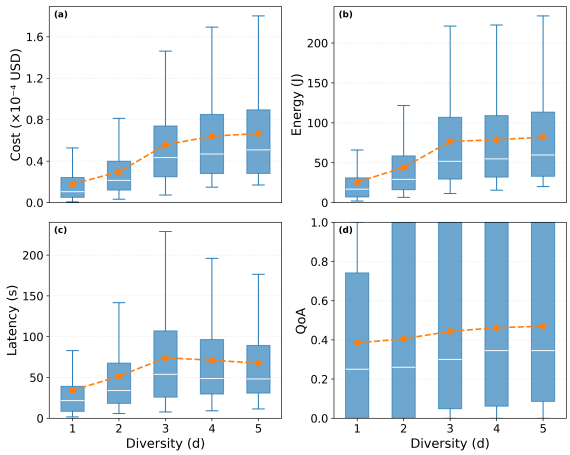


Figure 6: Distribution of QoA across diversity levels.

Figure 6 shows non-linear metric results effects. QoA improves with diversity, with the largest gain from d=2⟶3, after which the improvements taper off. Cost and energy increase substantially up to $d = 3$ and continue to increase modestly for $d > 3$. Latency peaks at $d = 3$ and declines thereafter (indicating diminishing QoA returns but improved time efficiency beyond a moderate level of diversity, $d = 3$). Marginal efficiency peaks at the d=2⟶3 transition for latency and at the d=3⟶4 for energy, suggesting that $d \in 3, 4$ balances QoA and resources. Dataset effect differ. On MMLU-Pro (knowledge-intensive), QoA generally increases with d but at higher resource cost: e.g., from $d = 1$ to $d = 5$, QoA increased in *Business* (0.44 ⟶ 0.54), *Computer Science* (0.30 ⟶ 0.64), *Physics* (0.22 ⟶ 0.28) and *Biology* showing the largest gain (0.80 ⟶ 0.96). At the same time, for instance, in *Biology* financial cost, latency, and energy increased 3.2×, 1.2×, and 2.4×, respectively. In contrast, SimpleQA's QoA is flat or declines with increasing d (e.g., *Art* 0.34⟶0.29), with only modest gains in a few categories (e.g., *TV Shows* 0.32⟶0.35, *Video Games* 0.31⟶0.42), that are often outpaced by cost/energy growth (> 3×). We next analyze sequential vs. hybrid/parallel plans performance against model diversity. For $d \geq 2$, hybrid/parallel plans consistently achieve a higher mean QoA than purely sequential plans (e.g., at $d = 2$, 0.463 vs. 0.384), but consume more resources

(higher latency and energy usage). Again, moderate diversity (d=3) is a robust operating point, maximizing QoA and with marginal efficiency; pushing to $d \in 4, 5$ increases financial cost/latency/energy without commensurate QoA gains.

Table 2: Incremental changes in QoA, Cost, Latency, and Energy for each step $d-1 \rightarrow d$.

Metric	Step (1→2)	Step (2→3)	Step (3→4)	Step (4→5)
ΔQoA	0.021	0.038	0.019	0.008
ΔCost (USD)	0.000	0.000	0.000	0.000
ΔLatency (s)	17.419	22.634	-2.995	-3.732
ΔEnergy (J)	18.373	32.601	2.099	3.287
ΔQoA/ΔCost (per $0.0001)	0.175	0.146	0.223	0.314
ΔQoA/ΔLatency (per s)	0.001	0.002	-0.006	-0.002
ΔQoA/ΔEnergy (per J)	0.001	0.001	0.009	0.002

Overall, model diversity enhances QoA up to $d = 3$, where hybrid and parallel plans achieve the best trade-off between quality and resource consumption. More diversity yields diminished QoA returns and higher cost, latency, and energy, particularly in knowledge-intensive datasets, showing the planner's ability to efficiently adapt the ensemble composition.

6.5 Impact of Budget

We evaluate execution under *hard* financial and latency constraints, leaving $E_{\max}$ and $QoA_{\min}$ unconstrained[9]. A plan π is acceptable iff $\text{Financial}(\pi) \le F_{\max}$ and $\text{Latency}(\pi) \le L_{\max}$; the planner may use up to K=5 operations. Unless noted otherwise, we use NSGA-II as the planning backend in this experiment, since it provides the best scalability–quality trade-off in our planner comparison. Budgets are instantiated at five levels by scaling model-averaged anchors ($0.000012, 19.47 s): $(F_{\max}(b),\ L_{\max}(b)) = b \cdot (\$0.000012,\ 19.47\,\text{s})$, $b \in \{1, \ldots, 5\}$. The anchor tuple specifies the mean observed financial cost and latency across the five distinct models, establishing a standardized baseline for resource allocation. Budgets are then generated by scaling this composite anchor, ensuring that each constraint is systematically derived from the aggregated typical resource usage of the system. In this setup, Opti-Q runs NSGA-II with feasibility dominance and prunes candidates using plan-time estimates of cost and QoA. For each selected plan, we label it as *budget-adherent* if budget constraints are met after its execution and as an *overrun* otherwise. For each of the five budgets, we report the *budget-adherence breakdown* (see Table 3) as well as the QoA on budget-adherent runs (see Figure 7).

The *adherence rate* remains high across budgets (96.6% at b=1 to 88.6% at b=5). Overruns are predominantly cost-driven (1.5–5.0%), with latency-only overruns at 0.0–3.3%, and joint cost+latency overruns at 0.6–4.7%. Mean QoA on budget-adherent selections shows a clear knee at b=3 (0.45), with only marginal changes thereafter (0.41 at b=4, 0.44 at b=5), indicating diminishing returns beyond moderate budgets. Dispersion increases with budget (IQR 0.31→0.52 from b=1→5), reflecting higher upside but greater variability under looser constraints. Budget utilization is conservative: budget-adherent plans consume 0.32 – 0.38 of the financial cost budget and 0.25–0.30 of the latency budget, indicating that the planner typically meets the targets without saturating them. Overall, the planner maintains high budget adherence across all levels, with overruns mainly cost-driven. QoA increases sharply at moderate budgets (b = 3) and then plateaus, while dispersion increases under looser budgets, indicating a trade-off between upside and variability.

[9]We budget only *Financial* and *Latency*—the primary levers for fast-and-cheap interactive QA; QoA is maximized (not budgeted) and Energy closely tracks latency. Focusing on $(F_{\max}, L_{\max})$ captures the user-facing trade-off while keeping the analysis simple.

Table 3: Budget-adherence breakdown per budget level.

b	Budget-adherence (%)	Overrun: Fin. (%)	Overrun: Lat. (%)	Overrun: Both (%)
1	96.6	2.8	0	0.6
2	91.4	2.6	3.3	2.7
3	95.8	1.5	1.2	1.5
4	88.8	5	2.3	3.9
5	88.6	4.3	2.5	4.6

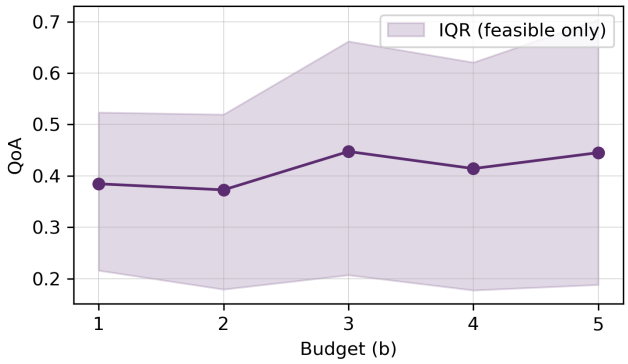


Figure 7: QoA vs. budget (*budget-adherent plans only*).

7 Conclusions and Future Work

We presented Opti-Q, a framework for optimizing multi-LLM QA via a multi-objective formulation. Opti-Q systematically constructs execution plans that balance QoA, cost, latency, and energy under explicit user-defined budget and constraints. Our formulation unifies sequential, parallel, and hybrid orchestration strategies within a single optimization space, enabling plan evaluation prior to execution. Opti-Q 's "pluggable optimization engine", guided by a statistical performance database efficiently estimates the cost of the plan and the QoA while exploring a large search space with a tractable overhead. Across QA benchmarks, Opti-Q consistently achieves higher accuracy at a comparable cost relative to baselines, demonstrating that structured optimization can outperform heuristic or static orchestration. In future work, we aim to explore mechanisms, such as caching previous results, to minimize the cost of planning while maximizing the estimated performance of the selected plans.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes,

Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. arXiv:2412.08905 [cs.CL] https://arxiv.org/abs/2412.08905

[2] Xavier Amatriain. 2024. Measuring and mitigating hallucinations in large language models: amultifaceted approach.

[3] Timothy W. Bickmore and Rosalind W. Picard. 2005. Establishing and maintaining long-term human-computer relationships. *ACM Trans. Comput. Hum. Interact.* 12 (2005), 293–327. https://api.semanticscholar.org/CorpusID:946518

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading Wikipedia to Answer Open-Domain Questions. arXiv:1704.00051 [cs.CL] https://arxiv.org/abs/1704.00051

[6] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv:2305.05176 [cs.LG] https://arxiv.org/abs/2305.05176

[7] Zhijun Chen, Jingzheng Li, Pengpeng Chen, Zhuoran Li, Kai Sun, Yuankai Luo, Qianren Mao, Dingqi Yang, Hailong Sun, and Philip S. Yu. 2025. Harnessing Multiple Large Language Models: A Survey on LLM Ensemble. arXiv:2502.18036 [cs.CL] https://arxiv.org/abs/2502.18036

[8] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. 2022. Scaling Instruction-Finetuned Language Models. arXiv:2210.11416 [cs.LG] https://arxiv.org/abs/2210.11416

[9] Jae-Won Chung, Jiachen Liu, Zhiyu Wu, Yuxuan Xia, and Mosharaf Chowdhury. 2023. ML.ENERGY Leaderboard. https://ml.energy/leaderboard.

[10] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. 2024. The rising costs of training frontier AI models. *arXiv preprint arXiv:2405.21015* n/a, n/a (2024), n/a.

[11] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[13] Prasenjit Dey, Srujana Merugu, and Sivaramakrishnan Kaveri. 2025. Uncertainty-Aware Fusion: An Ensemble Framework for Mitigating Hallucinations in Large Language Models. *arXiv preprint arXiv:2503.05757* n/a, n/a (2025), n/a.

[14] Thomas G. Dietterich. 2000. Ensemble Methods in Machine Learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems (MCS '00)*. Springer-Verlag, Berlin, Heidelberg, 1–15.

[15] Marko Durasevic, Francisco Javier Gil-Gala, and Domagoj Jakobović. 2023. Does size matter? On the influence of ensemble size on constructing ensembles of dispatching rules. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (Lisbon, Portugal) *(GECCO '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 559–562. doi:10.1145/3583133.3590562

[16] Shangbin Feng, Wenxuan Ding, Alisa Liu, Zifeng Wang, Weijia Shi, Yike Wang, Zejiang Shen, Xiaochuang Han, Hunter Lang, Chen-Yu Lee, Tomas Pfister, Yejin Choi, and Yulia Tsvetkov. 2025. When One LLM Drools, Multi-LLM Collaboration Rules. arXiv:2502.04506 [cs.CL] https://arxiv.org/abs/2502.04506

[17] Jianfeng Gao, Michel Galley, and Lihong Li. 2019. Neural Approaches to Conversational AI: Question Answering, Task-oriented Dialogues and Social Chatbots. doi:10.1561/1500000074

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[19] Maarten Grootendorst. 2022. BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *arXiv preprint arXiv:2203.05794* n/a, n/a (2022), n/a.

[20] Aamir Hamid, Hemanth Reddy Samidi, Tim Finin, Primal Pappachan, and Roberto Yus. 2023. GenAIPABench: A benchmark for generative AI-based privacy assistants. *arXiv preprint arXiv:2309.05138* n/a, n/a (2023), n/a.

[21] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 752–765. doi:10.14778/3503585.3503586

[22] Keke Huang, Yimin Shi, Dujian Ding, Yifei Li, Yang Fei, Laks Lakshmanan, and Xiaokui Xiao. 2025. ThriftLLM: On Cost-Effective Selection of Large Language Models for Classification Queries. *Proc. VLDB Endow.* 18, 11 (July 2025), 4410–4423. doi:10.14778/3749646.3749702

[23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[24] Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. LLM-Blender: Ensembling Large Language Models with Pairwise Ranking and Generative Fusion. arXiv:2306.02561 [cs.CL] https://arxiv.org/abs/2306.02561

[25] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. arXiv:2004.04906 [cs.CL] https://arxiv.org/abs/2004.04906

[26] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *The Twelfth International Conference on Learning Representations*.

[27] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[28] LangChain. 2024. *LangChain Documentation*. Placeholder Organization Name. https://python.langchain.com/docs/introduction/ Accessed: 2024-02-05.

[29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL] https://arxiv.org/abs/2005.11401

[30] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81. https://aclanthology.org/W04-1013/

[31] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)* (2025).

[32] Zihan Liu, Wei Ping, Rajarshi Roy, Peng Xu, Chankyu Lee, Mohammad Shoeybi, and Bryan Catanzaro. 2024. ChatQA: Surpassing GPT-4 on Conversational QA and RAG. *arXiv preprint arXiv:2401.10225* n/a, n/a (2024), n/a.

[33] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. doi:10.1145/3448016.3452838

[34] Leland McInnes, John Healy, Steve Astels, et al. 2017. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.* 2, 11 (2017), 205.

[35] Leland McInnes, John Healy, and James Melville. 2020. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. arXiv:1802.03426 [stat.ML] https://arxiv.org/abs/1802.03426

[36] OpenAI. 2025. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2026-02-20.

[37] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Pierre Isabelle, Eugene Charniak, and Dekang Lin (Eds.). Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. doi:10.3115/1073083.1073135

[38] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. arXiv:2204.05149 [cs.LG] https://arxiv.org/abs/2204.05149

[39] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115

[40] Matthew Russo, Sivaprasad Sudhir, Gerardo Vitagliano, Chunwei Liu, Tim Kraska, Samuel Madden, and Michael J. Cafarella. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *ArXiv* abs/2505.14661 (2025). https://api.semanticscholar.org/CorpusID:278768629

[41] Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2023. Querying Large Language Models with SQL. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:257913347

[42] Shivanshu Shekhar, Tanishq Dubey, Koyel Mukherjee, Apoorv Saxena, Atharv Tyagi, and Nishanth Kotla. 2024. Towards Optimizing the Costs of LLM Usage. arXiv:2402.01742 [cs.CL] https://arxiv.org/abs/2402.01742

[43] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Anna Korhonen, David Traum, and Lluís Màrquez (Eds.). Association for Computational Linguistics, Florence, Italy, 3645–3650. doi:10.18653/v1/P19-1355

[44] Shuhei Suzuoki and Keiko Hatano. 2024. Reducing hallucinations in large language models: A consensus voting approach using mixture of experts.

[45] Gemma Team. 2024. Gemma. *n/a* n/a, n/a (2024), n/a. doi:10.34740/KAGGLE/M/3301

[46] Selim Furkan Tekin, Fatih Ilhan, Tiansheng Huang, Sihao Hu, and Ling Liu. 2024. LLM-TOPLA: Efficient LLM Ensemble by Maximising Diversity. arXiv:2410.03953 [cs.CL] https://arxiv.org/abs/2410.03953

[47] Immanuel Trummer and Christoph Koch. 2014. Approximation schemes for many-objective query optimization. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1299–1310.

[48] Immanuel Trummer and Christoph Koch. 2016. A fast randomized algorithm for multi-objective query optimization. In *Proceedings of the 2016 International Conference on Management of Data*. 1737–1752.

[49] Jason Wei, Nguyen Karina, Hyung Won Chung, Yunxin Joy Jiao, Spencer Papay, Amelia Glaese, John Schulman, and William Fedus. 2024. Measuring short-form factuality in large language models. arXiv:2411.04368 [cs.CL] https://arxiv.org/abs/2411.04368

[50] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[51] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2025. A query optimization method utilizing large language models. *arXiv preprint arXiv:2503.06902* (2025).

[52] Fanghua Ye, Mingming Yang, Jianhui Pang, Longyue Wang, Derek Wong, Emine Yilmaz, Shuming Shi, and Zhaopeng Tu. 2024. Benchmarking llms via uncertainty quantification. *Advances in Neural Information Processing Systems* 37 (2024), 15356–15385.

[53] Jie You, Jaehoon Chung, and Mosharaf Chowdhury. 2022. Zeus: Understanding and Optimizing GPU Energy Consumption of DNN Training. *ArXiv* abs/2208.06102, n/a (2022), n/a. https://api.semanticscholar.org/CorpusID:251554526

[54] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1466–1479. doi:10.14778/3583140.3583160

A Appendix

A.1 Evolutionary Optimization with NSGA-II

This appendix provides the full pseudocode for the NSGA-II optimization procedure used to generate Pareto-optimal multi-LLM execution plans. Each candidate plan π represents a specific orchestration of LLMs with estimated cost, latency, energy, and QoA. The algorithm 1 iteratively evolves a population of such plans through selection, crossover, and mutation to approximate the Pareto front under user-defined constraints.

Algorithm 1 NSGA-II for Multi-Objective Schedule Optimization

Require: Initial population $\mathcal{P}_0$, population size $N = |\mathcal{P}_0|$, generations G, crossover probability C_r,
mutation operators $\mu \leftarrow \{\mu_1 : \text{edge mod.}, \mu_2 : \text{node mod.}, \mu_3 : \text{LLM adj.}\}$,
constraints ($F_{\max}, L_{\max}, E_{\max}, \text{QoA}_{\min}$), performance database PerfDB,
user preference vector $W = (w_F, w_L, w_E, w_{\text{QoA}})$
Ensure: Schedule selected from the first Pareto front based on W
for all $\pi_i \in \mathcal{P}_0$ **do** ▷ Initial population evaluation
  $(\text{QoA}_i, \text{Cost}_i, \text{Latency}_i, \text{Energy}_i) \leftarrow \text{QueryOrPredict}(\pi_i, \text{PerfDB})$
  if $\text{Cost}_i > F_{\max}$ **OR** $\text{Latency}_i > L_{\max}$ **OR** $\text{Energy}_i > E_{\max}$ **OR** $\text{QoA}_i < \text{QoA}_{\min}$ **then**
    Mark π_i as infeasible
  end if
  π_i.obj $\leftarrow (-\text{Cost}_i, -\text{Latency}_i, -\text{Energy}_i, \text{QoA}_i)$
end for
for $g \leftarrow 0$ **to** $G-1$ **do** ▷ Main evolutionary loop
  $\{\mathcal{F}_1, \mathcal{F}_2, \dots\} \leftarrow \text{FastNonDominatedSort}(\mathcal{P}_g)$
  $\text{CrowdingDistance}(\mathcal{F}_k)$ for all $\mathcal{F}_k$
  $\mathcal{M} \leftarrow \text{BinaryTournamentSelection}(\mathcal{P}_g, N)$
  $\mathcal{Q}_g \leftarrow \emptyset$
  while $|\mathcal{Q}_g| < N$ **do**
    $(S_p, S_q) \leftarrow \text{SelectParents}(\mathcal{M})$
    $(O_1, O_2) \leftarrow \begin{cases} \text{Crossover}(S_p, S_q) & \textbf{if } \text{rand}() < C_r \\ (S_p, S_q) & \textbf{otherwise} \end{cases}$
    $O_1 \leftarrow \text{Mutate}(O_1, \mu)$; $O_2 \leftarrow \text{Mutate}(O_2, \mu)$
    $\mathcal{Q}_g \leftarrow \mathcal{Q}_g \cup \{O_1, O_2\}$
  end while
  for all $\pi_i \in \mathcal{Q}_g$ **do**
    $(\text{QoA}_i, \text{Cost}_i, \text{Latency}_i, \text{Energy}_i) \leftarrow \text{QueryOrPredict}(\pi_i, \text{PerfDB})$
    if $\text{Cost}_i > F_{\max}$ **OR** $\text{Latency}_i > L_{\max}$ **OR** $\text{Energy}_i > E_{\max}$ **OR** $\text{QoA}_i < \text{QoA}_{\min}$ **then**
      Mark π_i as infeasible
    end if
    π_i.obj $\leftarrow (-\text{Cost}_i, -\text{Latency}_i, -\text{Energy}_i, \text{QoA}_i)$
  end for
  $\mathcal{R}_g \leftarrow \mathcal{P}_g \cup \mathcal{Q}_g$
  $\{\mathcal{F}_1, \mathcal{F}_2, \dots\} \leftarrow \text{FastNonDominatedSort}(\mathcal{R}_g)$
  $\mathcal{P}_{g+1} \leftarrow \text{SelectNextGeneration}(\{\mathcal{F}_k\}, N)$
end for
return $\text{SelectByPreference}(\text{GetFront}(\text{P}_G, 1), \text{W})$

A.2 Traversal Algorithm

The following procedure 2 describes the traversal logic used to evaluate execution plans in Opti-Q. It performs a breadth-first traversal over the DAG representation of an orchestration plan, aggregating token-based metrics (Cost, Latency, Energy, QoA) for each node while handling both sequential and parallel blending nodes. Matched and BlendRef objects are constructed as described in ProcessPlan.

We employ a modified BFS traversal to estimate the Cost, Energy, Latency, and *QoA* of a plan. ProcessPlan A.4 will return two mappings *Matched* and *BlendRef*, which respectively contain information pertaining to matched sub-plans and *QoA* information of the plan's parallel executions. The traversal will make use of these mappings to guide the estimation. There are certain Objects defined in the algorithm for the purposes of grouping essential stored information during traversal.

Algorithm 2 Traversal Procedure for Plan Evaluation

Require: DAG G, Query Type QT, *Matched*, *BlendRef*
Ensure: Estimated $(Cost, Latency, Energy, QoA)$
Object: Metrics → 4-tuple $(Cost, Latency, Energy, QoA)$
Object: ProcessingTableEntry → 2-tuple $(Node, Metrics)$
Object: QueueEntry → 3-tuple $(Node, Metrics, isStartNode)$
Object: ProcessingTable → Map $Node \mapsto$ ProcessingTableEntry
Initialize queue $Qu \leftarrow \emptyset$
Initialize map $PT \leftarrow \emptyset$
Initialize set $SeenMatchedSinks \leftarrow \emptyset$
$startNodes \leftarrow \text{GetNodesInDegree}(0, G)$
for all $L_i \in startNodes$ **do**
  $M \leftarrow (0, 0, 0, 0)$
  $isStartNode \leftarrow$ **True**
  $entry \leftarrow (L_i, M, isStartNode)$
  Qu.enqueue$(entry)$
end for
while Qu is not empty **do** ▷ Breadth-first traversal
  $(node, M, isStartNode) \leftarrow Qu$.dequeue()
  if $\text{isBlendNode}(node, G)$ **AND** $node \notin$ Matched **AND** $PT[node]$.length = Indegree$(node, G)$ **then**
    $M \leftarrow \text{GetParallelMetrics}(node, BlendRef, PT)$
  else
    Qu.enqueue$((node, M, isStartNode))$
    continue
  end if
  $successors \leftarrow \text{GetSuccessors}(G, node)$
  if $|successors| = 0$ **then**
    return M
  end if
  for all $s \in successors$ **do**
    if $s \in SeenMatchedSinks$ **then**
      continue
    end if
    if $s \in Matched$ **then**
      $M \leftarrow Matched[s]$
      $entry \leftarrow (s, M,$ **False**$)$
      Qu.enqueue$(entry)$
      continue
    end if
    $(M, QueueUp) \leftarrow \text{ProcessNode}(G, QT, c, s, M, PT)$
    if $QueueUp =$ **True then**
      Qu.enqueue$((s, M,$ **False**$))$
    end if
  end for
end while
return M

(1) **Metrics**
(2) **ProcessingTable**
(3) **ProcessingTableEntry**
(4) **QueueEntry**

Metrics is 4-Tuple containing the aggregated Cost, Latency, Energy, and estimated *QoA* during the BFS-traversal. For example at node L_i, there will be an aggregated amount for each metric stored in a **Metrics** object. When the traversal adds node L_j, a successor of L_i, to the BFS queue, during processing of L_j the aggregated metrics will be updated due to the influence of a L_j execution.

ProcessingTable is a map which maps Nodes to **ProcessingTableEntry** structures. A **ProcessingTableEntry** object is a 2-Tuple containing a node and a **Metrics** object. The purpose of ProcessingTableEntry is to store the aggregated metrics from all incoming branches to a blend node. With this, we can store all the information needed for parallel execution aggregation in an ordered fashion. In addition, we can track whether all necessary information has propagated to the blend node. For instance, say node L_i is a blend node

with nodes L_1, L_2, L_3, L_4 as its inputs. To properly estimate *QoA* and aggregate Cost, Latency, Energy, we need to wait for simulated execution of L_1, L_2, L_3, L_4 to finish. Hence, the **ProcessingTable** and its entries will indicate when it feasible to calculate an updated set of metrics simply by checking for the existence of an entry for the input nodes.

QueueEntry is a 3-Tuple containing a node, **Metrics** object, and a boolean flag indicating whether the entry is for a start node. For the traversal function, whether a node is a start node is an important distinction because it will affect how we propagate information.

The traversal will begin by entering all start nodes of the plan into the Queue as a **QueueEntry** object, which will initially contain 0 for all metrics in the associated **Metrics** object. We also initialize the **ProcessingTable** and a set to store seen nodes. Because of the DAG structure nodes with in-degree of 1 will only be vistited at most one time. For these types of node, we do not want to re-enter them into the queue, hence the set *SeenMatchedSinks*.

Once the inital state of the queue is initialized, processing of the entry is done as such. First, we pop a *QueueEntry* from the beginning of the queue, extracting the node, accumulated **Metrics**, and isStartNode flag.

Before performing BFS on the successors, we check if the current node is a blend node and is not in *Matched*. If that criteria is met, we are dealing with a parallel execution that is also not apart of a matched subplan. In this case, we need to aggregate all the metrics of each incoming branch, and estimate the *QoA* with the parallel *QoA* estimation method. For this, the information will be contained in the processing table. However, we only aggregate if information for incoming branches in the table for the blend node. Eventually in the midst of traversal, the information will propagate. However, if we reach the blend node before this occurs, then we simply readd the blend node back into the queue, and, then, continue to the next iteration. Readding back into the queue, will allow for reprocessing at a later time when the information will be available.

From here we proceed to analyzing the successors of the node in relation to its place in the DAG representation of the plan. Before processing each successor, we first check if the successor is the sink of a matched subplans contained in *Matched*. If so, we forget any accumulated Metrics aggregated at this point in the traversal, and, instead, and replace it with the metrics contained in *Matched* as these are metrics gathered from real-time fuzzy/exact executions. Then, for each successor a sub-procedure, ProcessNode, is called to determine how to aggregate the metrics and estimate the *QoA* with the current and successor Node. Because, the current and successor node can either be a blend or sequential node, there are rules followed for aggregation based on the node distinction. ProcessNode will aggregate Cost, Latency, and Energy and estimate *QoA* based on the type of nodes the current and successor are, sequential or blend. ProcessNode will then return the newly aggregated metrics and a boolean indicator indicating whether the successor node needs to be readded into the queue.

A.3 Size of the Acyclic Multi-LLM Plan Space

We derive a closed form for the number of distinct *labeled* acyclic orchestration plans of depth k over a model library L.

Setting. A plan consists of k ordered stages (nodes) indexed $1, \ldots, k$. Each stage selects a model from L (repetitions allowed). Information flows along directed edges that respect the order, i.e., an edge may exist only from i to j when $i < j$, ensuring acyclicity. Let Π denote the set of all such plans.

Proposition A.1 (Labeled acyclic plan count).

$$\left|\Pi\right| \;=\; |L|^k \;\cdot\; 2^{\binom{k}{2}} \;=\; |L|^k \;\cdot\; 2^{\frac{k(k-1)}{2}}.$$

Proof. *(Model assignments)* For each of the k ordered stages, choose any model in L with reuse permitted. By independence across stages, the number of stage-to-model assignments is

$$N_{\text{assign}} \;=\; |L|^k.$$

(Acyclic topologies) With the node order fixed as $1 < \cdots < k$, a directed edge may only go from i to j with $i < j$. There are exactly

$$N_{\text{edges}} \;=\; \binom{k}{2} \;=\; \frac{k(k-1)}{2}$$

eligible edges, each chosen independently to be present or absent. Hence the number of admissible upper-triangular (and therefore acyclic) edge sets is

$$N_{\text{conn}} \;=\; 2^{\binom{k}{2}} \;=\; 2^{\frac{k(k-1)}{2}}.$$

(Independence and product rule) Model choices and edge choices are independent design dimensions; multiplying yields

$$|\Pi| \;=\; N_{\text{assign}} \cdot N_{\text{conn}} \;=\; |L|^k \cdot 2^{\frac{k(k-1)}{2}}.$$

□

Remarks. (i) This is a labeled count with the execution order $1<\ldots<k$ fixed; it upper-bounds any implementation that later canonicalizes isomorphic topologies or imposes extra syntactic constraints (e.g., requiring a single sink or immediate blending at fan-ins). (ii) The expression makes the combinatorial growth explicit: $\log|\Pi| = k\log|L| + \frac{k(k-1)}{2}\log 2$, showing linear growth in k from labeling and quadratic growth from connectivity.

A.4 Tuning approximation parameter ϵ.

B Pruning Comparison:

C Planer Comparison:

No pruning		$\varepsilon = \Delta$	ε-dominance		Δ-QoA gate	
HV (%)	Time (s)		HV (%)	Time (s)	HV (%)	Time (s)
100.0	227.5	0.01	95.5	22.8	96.3	14.5
100.0	227.5	0.02	95.3	20.2	95.8	9.5
100.0	227.5	0.05	93.3	11.6	92.3	2.1
100.0	227.5	0.10	90.3	7.7	78.8	0.2
100.0	227.5	0.20	83.0	3.7	68.9	0.0

Table 4: Runtime–quality trade-off of DP pruning strategies ($k = 5$, averaged over 10 topics).

Table 5: Algorithm comparison across DAG sizes $k = 1$–5, averaged over 10 topics (3 runs for stochastic algorithms). HV% and IGD are computed w.r.t. the DP Pareto set when DP is tractable (DP is treated as the reference frontier). Best values per k are in bold (excluding the DP reference for HV% and IGD). ↓: lower is better, ↑: higher is better.

k	Algorithm	Time (s) ↓	HV% ↑	IGD ↓
	NSGA-II	1.3 ± 0.0	**100.0 ± 0.0**	**0.0000 ± 0.0000**
	DP	**0.0**	100.0	0.0000
1	DP+Δ	**0.0**	100.0	0.0000
	Hill Climbing	1.3 ± 0.0	**100.0 ± 0.0**	**0.0000 ± 0.0000**
	Hill Climbing+Δ	1.3 ± 0.0	100.0 ± 0.0	0.0136 ± 0.0000
	NSGA-II	3.4 ± 0.0	**100.0 ± 0.0**	**0.0000 ± 0.0000**
	DP	**0.0**	100.0	0.0000
2	DP+Δ	**0.0**	99.6	0.0953
	Hill Climbing	3.4 ± 0.0	**100.0 ± 0.0**	**0.0000 ± 0.0000**
	Hill Climbing+Δ	3.4 ± 0.0	99.3 ± 0.0	0.1907 ± 0.0000
	NSGA-II	10.2 ± 0.1	**100.0 ± 0.0**	**0.0000 ± 0.0000**
	DP	**0.4**	100.0	0.0000
3	DP+Δ	0.1	97.7	0.0955
	Hill Climbing	10.3 ± 0.0	99.7 ± 0.3	0.0130 ± 0.0076
	Hill Climbing+Δ	10.3 ± 0.0	97.3 ± 0.3	0.1690 ± 0.0334
	NSGA-II	10.1 ± 0.1	**100.0 ± 0.0**	**0.0001 ± 0.0000**
	DP	8.0	100.0	0.0000
4	DP+Δ	**0.4**	94.4	0.1191
	Hill Climbing	10.2 ± 0.0	93.2 ± 1.0	0.0824 ± 0.0159
	Hill Climbing+Δ	10.4 ± 0.3	91.2 ± 1.1	0.1575 ± 0.0285
	NSGA-II	21.0 ± 0.3	**99.9 ± 0.1**	**0.0043 ± 0.0016**
	DP	208.5	100.0	0.0000
5	DP+Δ	**1.9**	92.3	0.1123
	Hill Climbing	21.1 ± 0.1	78.4 ± 4.4	0.1078 ± 0.0175
	Hill Climbing+Δ	21.1 ± 0.1	76.9 ± 4.3	0.1352 ± 0.0247