LLM-QP: Query Planning for Large Language Model Inference

A Cost-Based Planner for Constrained Decoding Execution

Jake Lawrence[1]

[1]Independent Researcher

March 2026

Abstract

Large language model constrained decoding forces every token through a validity check whose cost depends on the size of the active token set. We observe that this creates an analogy to *query planning* in databases: multiple execution strategies (dense projection, sparse adjacency scoring, amortized updates) are semantically equivalent yet have markedly different runtime profiles. **LLM-QP** formalises this observation by defining plan equivalence, deriving roofline crossover conditions, and casting plan selection as a contextual bandit that learns online from latency and quality telemetry. We further show how the planner integrates naturally into MLIR / StableHLO compiler pipelines. Synthetic benchmarks confirm $\sim$10$\times$ cumulative cost savings from amortization and sub-linear router regret relative to an oracle policy.

Keywords: query planning, constrained decoding, contextual bandits, sparse inference, compiler integration, LLM systems

1 Introduction

Constrained decoding—generating text that conforms to a grammar, schema, or other structural specification—has become a core capability in LLM deployment. JSON mode, function calling, and structured output all rely on constraining the token set at each decode step to enforce validity. The dominant implementation strategy is *dense masked scoring*: compute logits over the full vocabulary V, mask invalid tokens to $-\infty$, and sample from the survivors.

This strategy is correct but wasteful. When a JSON schema restricts the next token to one of $K = 3$ valid continuations, a dense projection still computes scores for all $|V| \approx 128{,}000$ vocabulary entries. The ratio $K/|V|$ can be as low as 10^{-5}, yet the system pays full cost at every step.

We observe that this situation is structurally analogous to *query planning* in database systems. A SQL query can be executed via a full table scan or an index lookup; the results are identical, but the costs differ by orders of magnitude depending on selectivity. In constrained decoding, the "selectivity" is the branching factor $K = |\mathcal{A}(s)|$—the number of valid tokens at constraint state s—and the execution strategies are dense projection, sparse adjacency scoring, and amortized state reuse.

LLM-QP formalises this observation. We define plan equivalence (§2), derive roofline crossover conditions (§3), introduce an amortized query hypothesis (§4), cast plan selection as a contextual bandit (§5), and show how the planner integrates into MLIR/StableHLO compiler pipelines (§6).

Companion work. This paper is part of a research program on classification as infrastructure in intelligent systems. A companion paper [1] applies the same structural analysis to agent cognition, arguing that the taxonomy of cognitive modules determines the behavioral ceiling for LLM agents. A position paper [2] develops the shared theoretical framework.

2 Plan Equivalence

Two inference plans P_1, P_2 are *equivalent* if they produce identical token sequences under identical model and constraint semantics.

Definition 2.1 (ϵ-Equivalence)**.** *Two plans are ϵ-equivalent if logits differ by at most ϵ over the valid candidate set and produce identical decoded outputs.*

Lemma 2.1 (Dense–Sparse Equivalence)**.** *Let $\mathcal{A}(s)$ denote the valid token set at constraint state s.*

Dense masked scoring:

$$\tilde{\ell}(v) = \begin{cases} \sigma(h, v) & v \in \mathcal{A}(s) \\ -\infty & \textit{otherwise} \end{cases}$$

Sparse scoring:

$$\ell(v) = \sigma(h, v), \quad v \in \mathcal{A}(s)$$

Then

$$\arg\max_{v \in V} \tilde{\ell}(v) = \arg\max_{v \in \mathcal{A}(s)} \ell(v).$$

Proof. For any $v \notin \mathcal{A}(s)$, $\tilde{\ell}(v) = -\infty$, so such v cannot be the argmax. For $v \in \mathcal{A}(s)$, $\tilde{\ell}(v) = \sigma(h, v) = \ell(v)$. Therefore the argmax over V under $\tilde{\ell}$ equals the argmax over $\mathcal{A}(s)$ under ℓ. □

3 Roofline Regime Analysis

Dense and sparse heads differ in arithmetic intensity. The dense projection head performs

$$W_{\text{dense}} \approx 2\, d\, |V|$$

floating-point operations, while the sparse head requires only

$$W_{\text{sparse}} \approx 2\, d\, K,$$

where $K = |\mathcal{A}(s)|$ is the branching factor. Sparse scoring dominates whenever $K < |V|$.

In practice, sparse scoring also incurs overhead from index gathering, irregular memory access, and a refinement probability p_r for low-confidence predictions. The effective sparse cost is therefore:

$$C_{\text{sparse}}(K) = 2dK + C_{\text{gather}}(K) + p_r \cdot 2d|V|$$

where the last term accounts for fallback to dense scoring when the sparse result is uncertain. Figure 1 shows the crossover under our synthetic cost model.

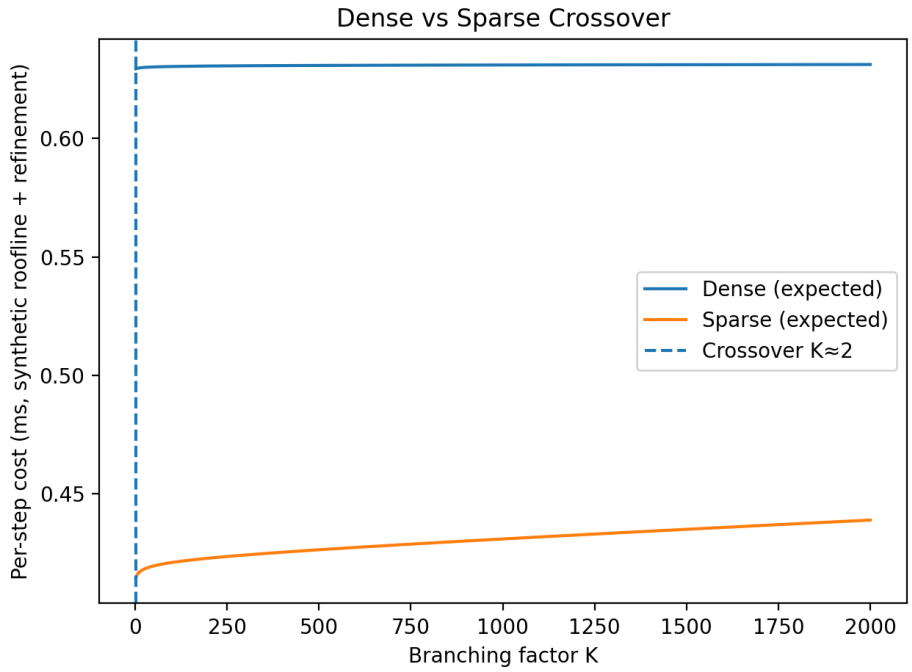

Figure 1: Dense vs. sparse per-step cost (synthetic roofline + refinement model). The dashed line marks the crossover at $K \approx 2$ in this cost regime.

4 Amortized Query Hypothesis

We decompose the decoder hidden state as

$$h_t = h^{\text{stable}} + \Delta h_t.$$

When the leading-token margin is large, full transformer recomputation contributes limited incremental information. This motivates an *amortized* execution strategy that reuses previous scores when confidence is high.

Margin. Define the scoring margin at step t as

$$m_t = \ell\Big(v^{(1)}\Big) - \ell\Big(v^{(2)}\Big).$$

Routing rule. Trigger a full refinement pass if $m_t < \tau$; otherwise, reuse the amortized score.

Figure 2 shows the cumulative cost comparison. Amortization yields approximately 10× savings over 4000 decode steps. The routing tradeoff frontier (Figure 3) shows the inverse relationship between amortized and full-recompute fractions.

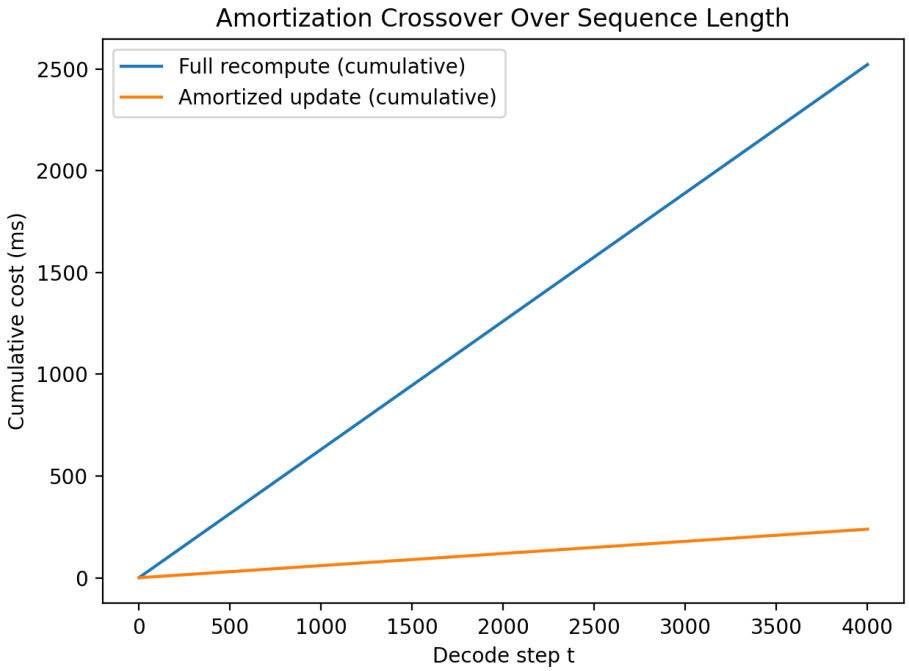

Figure 2: Cumulative cost: full recomputation vs. amortized update over decode length. Amortization yields $\sim 10\times$ savings by $t = 4000$.

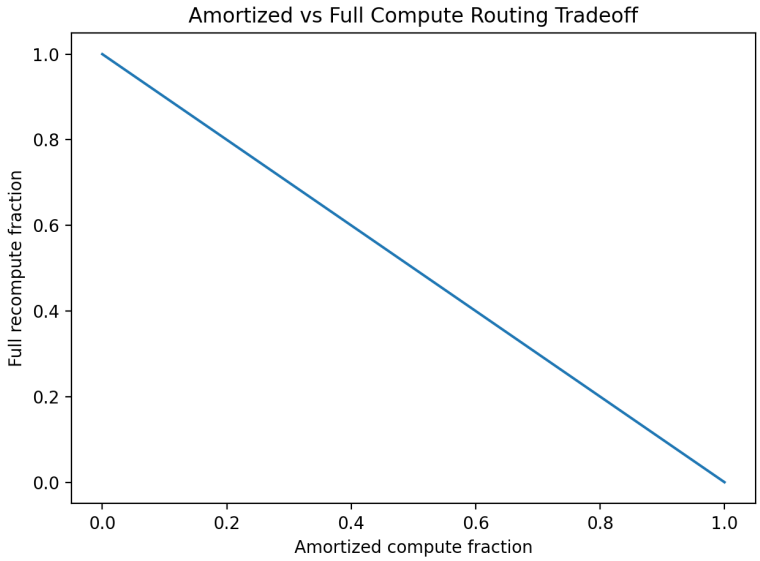

Figure 3: Amortized vs. full recompute routing tradeoff. The frontier is approximately linear under the synthetic cost model.

4.1 Margin Distribution

The margin m_t depends on the branching factor K. When the valid set is small, the leading candidate tends to dominate, producing large margins. As K grows, competition among candidates increases and margins shrink. Figure 4 confirms this inverse relationship in synthetic data.

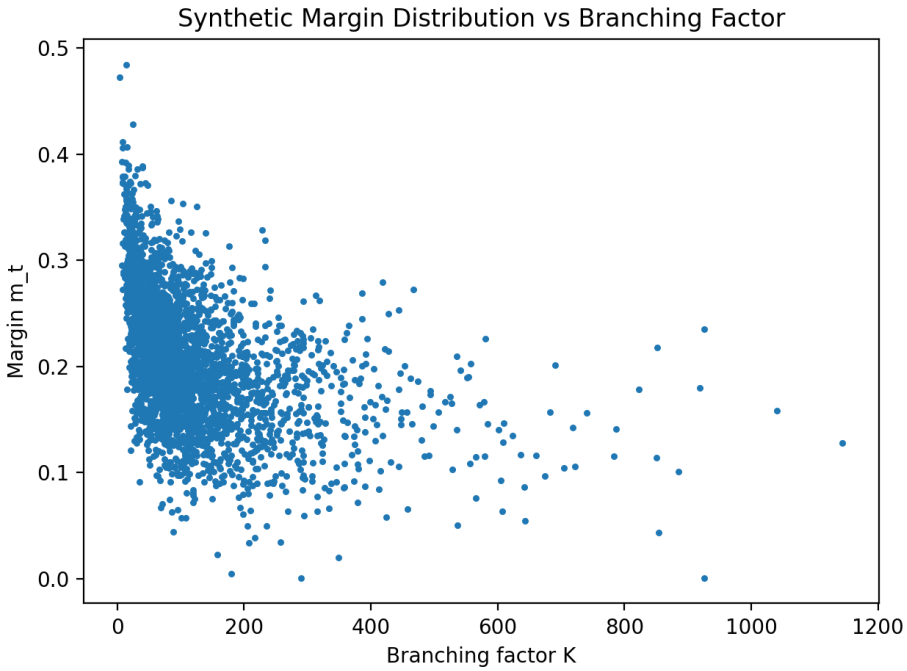

Figure 4: Synthetic margin distribution vs. branching factor K. Margins decay as the candidate set grows, motivating adaptive refinement.

4.2 Threshold Sensitivity

The routing threshold τ controls the tradeoff between amortization savings and unnecessary refinement cost. Figure 5 shows the average routing cost as a function of τ.

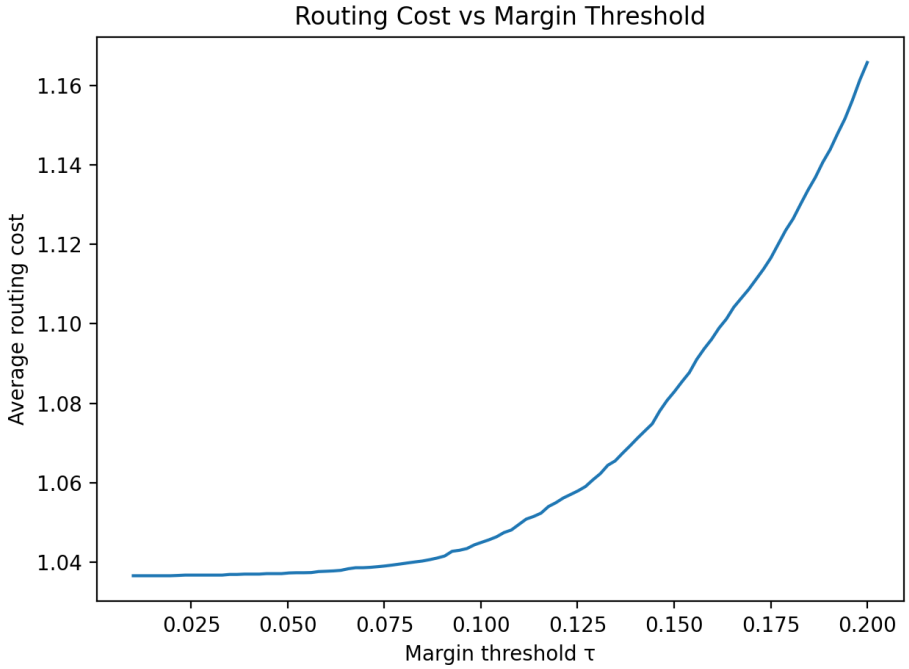

Figure 5: Average routing cost vs. margin threshold τ. Below $\tau \approx 0.075$ the cost is nearly flat; beyond that, unnecessary refinements dominate.

5 Planner Routing as a Contextual Bandit

At decoding step t the planner observes context $x_t \in \mathcal{X}$ and selects an execution plan $a_t \in \mathcal{A}$. The scalar loss combines latency and quality:

$$\ell_t(a) = \lambda \cdot \mathrm{Latency}_t(a) + (1-\lambda) \cdot \mathrm{QualityLoss}_t(a).$$

5.1 Oracle and Regret

The per-step oracle is $a_t^* = \arg\min_{a \in \mathcal{A}} \ell_t(a)$, yielding **dynamic regret**

$$R_T^{\mathrm{dyn}} = \sum_{t=1}^{T} \ell_t(a_t) - \sum_{t=1}^{T} \ell_t(a_t^*).$$

For stationary regimes the **static regret** is

$$R_T^{\mathrm{stat}} = \sum_{t=1}^{T} \ell_t(a_t) - \min_{a \in \mathcal{A}} \sum_{t=1}^{T} \ell_t(a).$$

5.2 Linear Contextual Bandit

Under realizability $\mathbb{E}[\ell_t(a) \mid x_t] = x_t^\top \theta_a$ with $\|x_t\| \le 1$, LinUCB / linear Thompson sampling achieves

$$R_T^{\mathrm{stat}} = \tilde{O}\Big(d\sqrt{T|\mathcal{A}|}\Big),$$

where $d = \dim(x_t)$ and $\tilde{O}$ hides logarithmic factors.

5.3 Nonstationarity and Variation Budget

To capture policy churn, define a variation budget

$$V_T = \sum_{t=2}^{T} \sup_{a \in \mathcal{A}} \big|\mathbb{E}[\ell_t(a) \mid x_t] - \mathbb{E}[\ell_{t-1}(a) \mid x_{t-1}]\big|.$$

Sliding-window or restart-based bandits achieve

$$R_T^{\mathrm{dyn}} = \tilde{O}\Big(\sqrt{T|\mathcal{A}|} + V_T^{1/3} T^{2/3}\Big)$$

under standard boundedness assumptions.

5.4 Connection to Margin-Based Routing

The threshold router π_τ from §4 forms a *finite policy class* when τ is discretised. Expert-style learning over $\{\pi_{\tau_i}\}$ gives regret $O\Big(\sqrt{T \log |\Pi|}\Big)$ for $|\Pi|$ candidate thresholds. More generally the planner can learn directly over actions via contextual features (branching factor K_t, depth, margin proxies, churn) and a linear bandit.

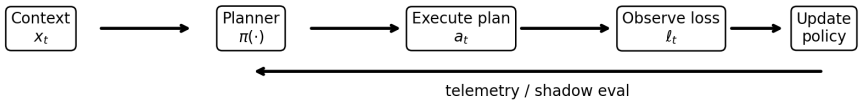

Figure 6: Bandit loop for LLM-QP. The planner maps runtime context to a plan, observes latency and quality proxies (optionally via shadow evaluation), and updates the routing policy online.

5.5 Synthetic Regret

Figure 7 shows cumulative regret of a threshold router against the oracle policy. Regret grows sub-linearly, consistent with the $\tilde{O}(\sqrt{T})$ bounds.

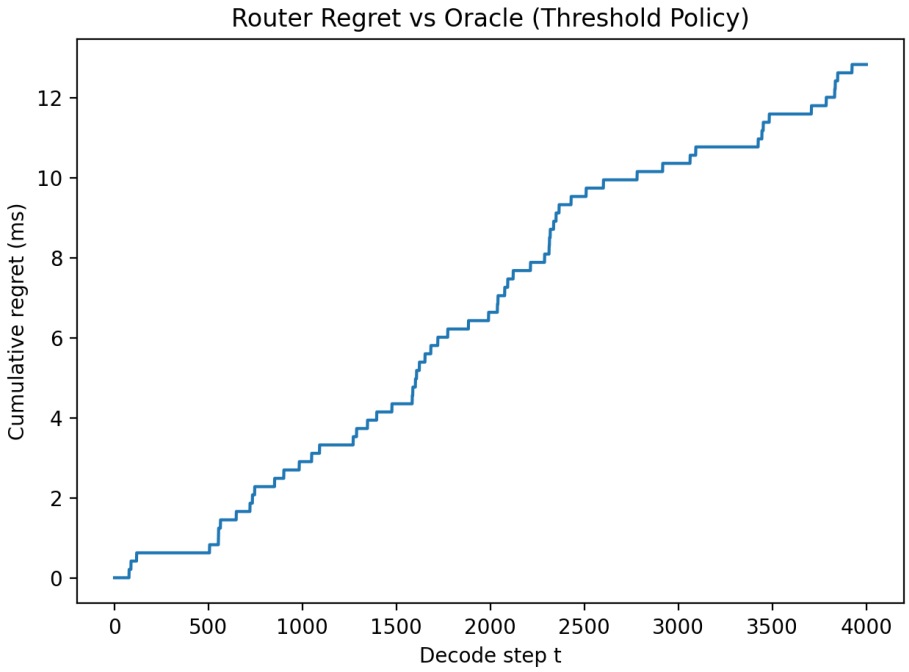

Figure 7: Cumulative router regret vs. oracle (threshold policy). Regret grows sub-linearly, confirming the bandit converges.

6 Compiler Integration: MLIR / StableHLO

Modern inference stacks compile model graphs through MLIR and StableHLO. LLM-QP can therefore be implemented as a *compiler pass* that introduces multiple equivalent execution plans and selects among them via a cost model.

6.1 Logical vs. Physical Plans

We define a single logical operator `DecodeStep(query, constraint_state)` expressing the semantics of one constrained decoding step. Physical implementations include:

1. Dense projection head
2. Sparse adjacency scoring
3. Amortized query update
4. Amortized update with rerank
5. Full recomputation (refinement)

Each produces equivalent outputs under the rules of §2.

6.2 Plan Expansion Pass

During compilation the planner expands the logical node into candidate physical plans:

```
DecodeStep
  -> DenseHead
  -> SparseHead
  -> AmortizedHead
  -> AmortizedHead + Rerank
```

These alternatives are represented as parallel subgraphs in MLIR.

6.3 Cost Model

Each plan is annotated with an estimated cost comprising kernel latency, memory bandwidth, refinement probability, and device utilisation. The aggregate cost follows the same model used by the bandit:

$$C = \lambda \cdot \text{Latency} + (1 - \lambda) \cdot \text{QualityLoss}.$$

Costs are evaluated either statically at compile time or dynamically via runtime telemetry.

6.4 Rewrite Rule

A rewrite pass selects the lowest-cost implementation:

```
if cost(SparseHead) < cost(DenseHead):
    use SparseHead
else:
    use DenseHead
```

In practice this is implemented using MLIR pattern rewrites or XLA custom calls.

6.5 Runtime Adaptation

Runtime telemetry (margins, branching factor, cache state) feeds back into the cost model. The compiler emits a small routing kernel that chooses the implementation at runtime. This hybrid compile-time / runtime planning architecture reuses existing compiler infrastructure while enabling fully adaptive execution.

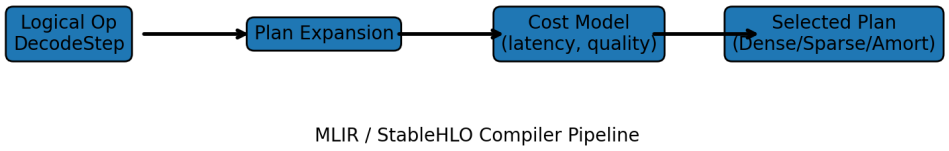

Figure 8: MLIR / StableHLO compiler pipeline. A logical `DecodeStep` is expanded into candidate physical plans; the cost model selects the cheapest at compile or run time.

7 End-to-End System Architecture

Figure 9 summarises the complete LLM-QP system.

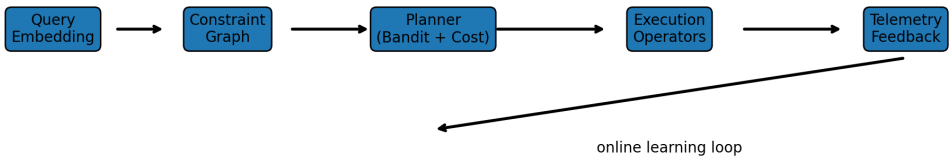

Figure 9: End-to-end architecture of LLM-QP. A constrained decoding query is transformed into a constraint graph, analysed by the planner, routed through the execution operator space, and refined via telemetry feedback.

The pipeline operates as follows:

1. **Query embedding** produces a representation of the decoding state.
2. The **constraint graph** defines valid token transitions.
3. The **planner** selects an execution plan using contextual signals.
4. The chosen **operator** executes the decode step.
5. **Telemetry** feeds back to update routing decisions.

This closed-loop architecture enables the system to adapt execution plans to runtime characteristics while maintaining equivalence guarantees.

8 Discussion

8.1 Relation to Database Query Optimisers

The analogy between constrained decoding and query planning is intentional and structural, not merely metaphorical. Database query optimisers choose between physical operators (hash join vs. nested loop, index scan vs. sequential scan) based on cost models that estimate I/O, CPU, and memory usage given data statistics. LLM-QP makes the same move: given statistics about the constraint state (branching factor K, margin m_t, decode depth), it selects among physical operators (dense, sparse, amortized) that produce equivalent results at different costs.

The key difference is that database statistics are typically gathered offline (via `ANALYZE`), while LLM-QP must estimate costs online during decoding. This motivates the bandit formulation: the planner learns the cost model from execution telemetry rather than from pre-computed statistics.

8.2 Limitations

1. **Synthetic evaluation.** All benchmarks use cost models rather than actual GPU timings. The crossover points and regret curves are illustrative, not empirical.

2. **Amortization hypothesis.** The claim that large margins permit score reuse is stated but not validated on real model hidden states.

3. **Single-device assumption.** The roofline analysis assumes a single accelerator. Tensor-parallel and pipeline-parallel inference introduces additional communication costs that may shift the crossover.

4. **No quality measurement.** The bandit's quality loss term $\text{QualityLoss}_t(a)$ is modelled synthetically. Measuring actual output degradation from amortized scoring on downstream tasks is critical future work.

8.3 Classification as Infrastructure

LLM-QP's plan lattice—the set of physical implementations considered by the planner—is not a neutral description of the execution space. It is a taxonomic commitment that determines what optimisations are representable. An optimiser whose plan lattice lacks an amortized operator cannot discover amortized savings, regardless of how sophisticated its cost model is. The lattice is not scaffolding; it is load-bearing structure.

This observation connects LLM-QP to a broader thesis developed in a companion position paper [2]: in any system where an agent must act under uncertainty, the classification layer functions as infrastructure—embedded, transparent when working, constitutive rather than descriptive.

9 Future Work

Several extensions are planned:

Hardware benchmarks. Profiling dense, sparse, and amortized kernels on real accelerators (A100, H100, TPU v5) to validate the synthetic cost models and identify actual crossover points.

Quality evaluation. Measuring downstream task accuracy under amortized scoring on benchmarks such as structured data extraction and grammar-constrained generation.

Adaptive bandits. Implementing the sliding-window bandit for nonstationary regimes and evaluating dynamic regret on real decoding traces.

Multi-device planning. Extending the cost model to tensor-parallel inference, where communication costs and load imbalance affect the dense/sparse tradeoff.

Integration with existing frameworks. Implementing LLM-QP as a plugin for outlines, guidance, and other constrained decoding libraries.

10 Conclusion

LLM-QP reframes constrained decoding as a query planning problem. By formalising plan equivalence, deriving roofline crossover conditions, and casting plan selection as a contextual bandit, it provides a principled framework for adaptive execution that goes beyond static heuristics. The compiler integration via MLIR/StableHLO demonstrates that the approach is deployable within existing infrastructure.

The central insight is that constrained decoding has variable selectivity—sometimes the valid set is nearly the full vocabulary, sometimes it is a single token—and a system that uses the same execution strategy regardless of selectivity leaves substantial performance on the table. LLM-QP fills this gap with the same cost-based reasoning that has proven effective in database systems for decades.

A Planner Pseudocode

A.1 LinUCB for Plan Selection

Let $A = |\mathcal{A}|$ and feature dimension d. Maintain per-action matrices $V_a \in \mathbb{R}^{d\times d}$ and vectors $b_a \in \mathbb{R}^d$, initialised as $V_a = \alpha I$, $b_a = 0$.

for $t = 1, 2, \ldots$ **do**
  Observe context x_t.
  for each action a **do**
    $\hat{\theta}_a \leftarrow V_a^{-1}\, b_a$
    $\mathrm{ucb}_a \leftarrow x_t^\top \hat{\theta}_a + \beta\sqrt{x_t^\top V_a^{-1} x_t}$
  end for
  $a_t \leftarrow \arg\min_a \ \mathrm{ucb}_a$
  Execute a_t; observe loss $\ell_t(a_t)$.
  $V_{a_t} \leftarrow V_{a_t} + x_t x_t^\top; \quad b_{a_t} \leftarrow b_{a_t} + x_t\, \ell_t(a_t)$.
end for

A.2 Expert-Over-Thresholds Routing

Discretise thresholds $\{\tau_1, \ldots, \tau_N\}$ and define policies $\pi_i = \pi_{\tau_i}$. Maintain weights w_i initialised uniformly.

for $t = 1, 2, \ldots$ **do**
  Observe context x_t.
  Compute each expert's action $a_t^{(i)} = \pi_i(x_t)$.
  Sample expert i proportional to w_i *(or pick* $\arg\max w_i$ *for greedy).*
  Execute $a_t = a_t^{(i)}$; observe loss $\ell_t(a_t)$.
  Update: $w_i \leftarrow w_i \exp\!\big(-\eta\, \hat{\ell}_{t,i}\big)$, where $\hat{\ell}_{t,i}$ is importance-weighted.

7: **end for**

This realises regret $O\big(\sqrt{T \log N}\big)$ against the best threshold policy in hindsight.

B Physical Plan Summary

Table 1: Physical execution plans and their cost characteristics.

Plan	Compute	When Preferred	Quality Risk
Dense head	$O(d\|V\|)$	$K \approx \|V\|$	None
Sparse head	$O(dK)$	$K \ll \|V\|$	None (exact)
Amortized update	$O(1)$	$m_t \gg \tau$	Low
Amortized + rerank	$O(dK)$	m_t moderate	Minimal
Full recompute	$O(d\|V\|)$	$m_t < \tau$ (refinement)	None

References

[1] Lawrence, J. (2026a). SAGEN: Situational awareness for generative agents—a modular cognitive architecture for persistent state in language model agent systems. *arXiv preprint*.

[2] Lawrence, J. (2026b). Classification is infrastructure: Planning layers, taxonomic commitment, and the architecture of intelligent systems. *arXiv preprint*.