Convex Resource Allocation for Efficient LLM Inference

Mathematical Formulation: From Discrete Scheduling to Convex Optimization

Satyam Srivastava Adyasha Patra Jay Chaudhary Keshav Gupta

1 Task Assignments

- **Satyam Srivastava:** Formulation of Primal/Dual problems and derivation of analytic KKT conditions to prove optimality of greedy strategies.
- **Adyasha Patra:** Implementation of the convex solver simulation using CVXPY and analysis of the duality gap/slack variables.
- **Jay Chaudhary:** Literature review (vLLM, AlpaServe) and implementation of baseline algorithms (like FCFS) for performance benchmarking.
- **Keshav Gupta:** Development of the "Fluid Flow" approximation model for continuous batch scheduling and visualization of throughput trade-offs.

2 Introduction

2.1 Motivation

Large Language Model (LLM) inference is fundamentally a *sequential token-generation* process. Given a user prompt, the model first processes the entire input in a **prefill phase** (computing the initial Key-Value cache), then enters a **decode phase** where it generates output tokens one at a time, each requiring a read from and a write to the KV cache. The KV cache memory footprint therefore grows linearly with the number of generated tokens.

A serving system receives many concurrent requests, each at a different stage of completion, and must decide *which requests to include in each batch*. This decision is constrained by two scarce resources:

(i) **Compute capacity**: the GPU can process at most a fixed number of tokens per forward pass.

(ii) **Memory capacity**: the total KV cache across all active requests must fit in GPU memory.

The memory constraint is what makes LLM scheduling fundamentally different from classical job scheduling: a request's resource consumption *grows over time* as it generates more tokens. Deciding to process a request today incurs a memory cost that persists into the future.

Our goal is to formulate this scheduling problem as a **convex optimization problem**, analyze its optimality conditions, and use the resulting structural insights to design practical scheduling policies.

2.2 Previous Works

Continuous Batching and Memory Management. Yu et al. [2] introduced *iteration-level scheduling* in Orca, replacing static batching with per-iteration admission and eviction of requests, achieving up to 36.9× throughput gains over FasterTransformer. Kwon et al. [1] proposed PagedAttention in vLLM, applying OS-style virtual memory paging to KV cache and reducing memory waste to under 4%. Agrawal et al. [7] addressed prefill–decode interference in Sarathi via chunked prefills that let decode tokens piggyback on compute-intensive prefill chunks. These systems optimize the *mechanism* of batching and *how* memory is managed, but schedule requests via FCFS and do not reason about *which* requests to prioritize. Our work fills this gap: we formulate the within-batch admission decision as a convex program whose dual variables λ_t (compute price) and μ_t (memory price) provide per-timestep priority signals that FCFS lacks.

Fluid-Guided Scheduling. The closest work to ours is Ao et al. [3], who also relax discrete token generation into continuous flows under KV cache constraints. Their approach is rooted in *queueing theory*: steady-state balance equations yield threshold-based algorithms (WAIT, Nested WAIT) proven asymptotically optimal via stochastic coupling. We approach the same problem through *convex optimization*—an explicit utility maximization analyzed via KKT conditions. The two are complementary: their convergence proofs validate our fluid relaxation, while our stationarity condition $U_i'(x_{i,t}^*) = \lambda_t^* + m_{\text{block}} \sum_{s=t}^{T} \mu_s^*$ reveals the resource-pricing structure that explains *why* their thresholds work. Our formulation additionally accommodates general concave utilities (α-fairness, proportional fairness) and non-stationary arrivals, whereas WAIT is specialized to throughput under Poisson arrivals. A shared limitation is the assumption of known output lengths; relaxing this via Bayesian estimation or receding-horizon re-optimization is future work.

Optimization Foundations. Our formulation adapts network utility maximization (NUM) [8] from communication networks to LLM scheduling. The key novelty is the *temporal coupling* in the memory constraint: a token generated at time t occupies KV cache at all future times, yielding a cumulative future memory price $m_{\text{block}} \sum_{s=t}^{T} \mu_s$ absent in static NUM settings. Standard convex optimization theory [5] (KKT conditions, strong duality) provides the analytical foundation.

2.3 Intended Contributions

This work makes the following contributions:

1. **Convex Reformulation of LLM Scheduling.** We formulate the LLM batch-scheduling problem as a convex program by relaxing binary decode decisions to continuous processing rates. The resulting primal problem has a concave objective and linear constraints, admitting efficient solution via standard solvers (CVXPY/Clarabel, ECOS, SCS).

2. **Dual Variable Analysis and Resource Pricing.** We derive the Lagrangian dual and show that strong duality holds under Slater's condition. The dual variables λ_t (compute price) and μ_t (memory price) provide interpretable, per-timestep signals that quantify resource bottlenecks—signals absent from heuristic schedulers such as FCFS.

3. **KKT-Based Structural Insights.** Via the KKT stationarity condition, we prove that the optimal policy exhibits *Shortest Remaining Job First* (SRJF) behavior under memory pressure and *max-min fair sharing* under compute pressure. These results give a principled, optimization-theoretic explanation for scheduling heuristics previously justified only empirically.

4. **Regime Classification Framework.** Using complementary slackness, we identify four operating regimes (underloaded, compute-bound, memory-bound, doubly constrained) and characterize the optimal policy in each. The dual variables serve as online diagnostic signals for regime detection.

5. **Empirical Validation.** We implement the full pipeline—fluid relaxation, convex solving, greedy discrete rounding, and FCFS/SJF baselines—and evaluate across six controlled scenarios on $N = 50$ synthetic requests. The convex-guided policy achieves $\sim$6% higher throughput than FCFS and SJF in constrained regimes while completing all requests, validating the theoretical predictions.

2.4 Organization of the Paper

The remainder of the paper is organized as follows. Section 3 formally defines the scheduling problem, introduces the fluid-flow relaxation, and states the primal convex program together with its dual. Section 3.5 derives the KKT optimality conditions and analyzes the four operating regimes that arise from complementary slackness, establishing the SJF and fair-sharing structural results. Section 4 describes the implementation pipeline, including the convex solvers used (Clarabel, ECOS, SCS), the greedy discrete-rounding procedure, and the FCFS and SJF baselines. Section 4.7 presents experimental results across six constraint scenarios, reports duality gaps, solver performance, and throughput comparisons, and discusses the theoretical limitation of the fluid relaxation in the memory-bound regime. Section 7 summarizes our findings and outlines two directions for future work: integration with production serving engines and handling unknown output lengths.

3 Statement of the Problem

We consider a single-GPU serving system operating over a discrete time horizon $t \in \{1, 2, \ldots, T\}$. Each time step corresponds to one decode iteration (one forward pass of the model).

Definition 3.1 (Request)**.** *A request i is characterized by:*

- L_i^{pre}*: the prefill length (number of input tokens), which determines the initial KV cache size after the prefill phase.*
- L_i^{dec}*: the total number of decode tokens to be generated (output length).*
- a_i*: the arrival time of request i.*

Definition 3.2 (Active Set)**.** *At time step t, the **active set** $\mathcal{R}_t$ denotes the collection of all requests that have arrived ($a_i \leq t$) and have not yet completed all their decode tokens.*

We define the following system parameters:

3.1 The Discrete Scheduling Problem

In reality, the scheduler makes *binary* decisions at each time step: each request either generates exactly one token (is included in the batch) or generates zero tokens (is excluded).

Definition 3.3 (Discrete Decision Variable)**.** *Let $z_{i,t} \in \{0, 1\}$ be the indicator variable:*

$$z_{i,t} = \begin{cases} 1 & \text{if request } i \text{ is scheduled for decoding at time } t, \\ 0 & \text{otherwise.} \end{cases}$$

Symbol	Description
C_{compute}	Maximum number of tokens the GPU can process per iteration
M_{GPU}	Total GPU memory available for KV cache (in units of cache blocks)
m_{block}	Memory consumed by one KV cache block (one token's key-value pair)
T	Total number of time steps (planning horizon)
$x_{i,t}$	Decision variable: processing rate of request i at time t
$U_i(\cdot)$	Utility function for request i (assumed concave and increasing)
$\mathcal{R}_t$	Active set at time t (requests that have arrived but not yet completed)

Table 1: Summary of notation.

The discrete scheduling problem is:

Discrete Problem (Integer Program)

$$\max_{z_{i,t}\in\{0,1\}} \quad \sum_{t=1}^{T} \sum_{i\in\mathcal{R}_t} U_i(z_{i,t}) \tag{1}$$

$$\text{subject to} \quad \sum_{i\in\mathcal{R}_t} z_{i,t} \le C_{\text{compute}}, \quad \forall\, t \in [T] \tag{2}$$

$$\sum_{i\in\mathcal{R}_t} \Big(L_i^{\text{pre}} + \sum_{\tau=1}^{t} z_{i,\tau} \Big) \cdot m_{\text{block}} \le M_{\text{GPU}}, \quad \forall\, t \in [T] \tag{3}$$

$$\sum_{\tau=1}^{T} z_{i,\tau} \le L_i^{\text{dec}}, \quad \forall\, i \tag{4}$$

$$z_{i,t} \in \{0,1\}, \quad \forall\, i, t \tag{5}$$

Interpretation of each constraint.

- **Constraint** (2) **(Compute Capacity):** At each time step, the total number of requests being decoded cannot exceed the GPU's compute throughput C_{compute}. This is a per-iteration batch-size limit.
- **Constraint** (3) **(Memory Budget):** The total KV cache memory at time t is the sum, over all active requests, of each request's current cache size. For request i, the cache size at time t is L_i^{pre} (from prefill) plus $\sum_{\tau=1}^{t} z_{i,\tau}$ (the number of decode tokens generated so far), multiplied by m_{block} (memory per token). This total must not exceed M_{GPU}.

  This constraint is the key challenge: it couples decisions across time, because a token generated at time $\tau < t$ continues to occupy memory at all future times until the request completes.
- **Constraint** (4) **(Completion):** Each request generates at most L_i^{dec} output tokens in total.

- **Constraint** (5) **(Integrality):** Decisions are binary—a request either decodes one token or does not.

Remark 3.1 (Computational Hardness)**.** *The discrete problem* (1)–(5) *is a variant of the multi-dimensional knapsack problem with temporal coupling. Even without the memory constraint, the problem is NP-hard in general. The temporal coupling in* (3) *makes it significantly harder, as the feasibility of a decision at time t depends on the entire history of decisions $\{z_{i,\tau}\}_{\tau<t}$.*

3.2 The Fluid Flow Relaxation

To make the problem tractable, we apply a **fluid flow relaxation**: instead of binary "decode or not" decisions, we allow each request a *continuous processing rate*.

3.2.1 Motivation for the Relaxation

The key insight is that in a large-scale serving system with hundreds of concurrent requests and scheduling decisions at millisecond granularity, the aggregate behavior of the scheduler is well-approximated by continuous flows. This idea is standard in queueing theory and network optimization [5, 3].

The continuous rate $x_{i,t} \in [0, 1]$ admits two interpretations:

(a) **Time-sharing**: Over a window of K decode iterations, $x_{i,t} = 0.5$ means request i is included in roughly half the batches. The continuous variable represents a scheduling fraction.

(b) **Fluid approximation**: When many requests flow through the system, discrete per-token decisions average out and the system's behavior looks smooth. The fluid model captures this aggregate behavior.

3.2.2 The Relaxed Decision Variable

Definition 3.4 (Continuous Processing Rate)**.** *Let $x_{i,t} \in [0, 1]$ represent the **continuous processing rate** of request i at time t. The value $x_{i,t}$ can be interpreted as the fraction of compute allocated to request i during time step t, or equivalently, the expected number of tokens generated by request i at time t.*

Under this relaxation:

- The **cumulative tokens generated** by request i up to time t is:

$$s_{i,t} \;=\; \sum_{\tau=1}^{t} x_{i,\tau} \;\in\; \mathbb{R}_{\geq 0}$$

  (a continuous quantity, rather than an integer count).
- The **KV cache size** of request i at time t becomes:

$$\mathrm{Cache}_i(t) \;=\; \left(L_i^{\mathrm{pre}} + s_{i,t}\right) \cdot m_{\mathrm{block}}$$

  This is now a smooth, monotonically increasing function of t (rather than a staircase).

3.3 The Convex Optimization Problem (Primal)

Replacing the binary $z_{i,t}$ with continuous $x_{i,t}$ and relaxing the integrality constraint, we obtain the following convex program.

3.3.1 Choice of Utility Function

We assume each request has a concave, non-decreasing utility function $U_i : [0, 1] \to \mathbb{R}$. The concavity of U_i encodes a preference for *fairness*: the marginal value of additional processing rate diminishes. Standard choices include:

- **Proportional fairness:** $U_i(x) = w_i \log(x)$, where $w_i > 0$ is a priority weight.
- **Weighted linear:** $U_i(x) = w_i\, x$ (reduces to throughput maximization).
- **α-fair utility:** $U_i(x) = w_i \frac{x^{1-\alpha}}{1-\alpha}$ for $\alpha > 0, \alpha \neq 1$.

Since U_i is concave, $-U_i$ is convex. Maximizing $\sum U_i$ is equivalent to minimizing $\sum(-U_i)$, which is a convex minimization problem.

3.3.2 Primal Formulation

Primal Problem (Convex Program – Maximization Form)

$$\max_{x_{i,t}} \quad \sum_{t=1}^{T} \sum_{i \in \mathcal{R}_t} U_i(x_{i,t}) \tag{6}$$

$$\text{subject to} \quad \sum_{i \in \mathcal{R}_t} x_{i,t} \leq C_{\text{compute}}, \quad \forall\, t \in [T] \qquad \text{(C1: Compute)}$$

$$\sum_{i \in \mathcal{R}_t} \left(L_i^{\text{pre}} + \sum_{\tau=1}^{t} x_{i,\tau} \right) \cdot m_{\text{block}} \leq M_{\text{GPU}}, \quad \forall\, t \in [T] \qquad \text{(C2: Memory)}$$

$$0 \leq x_{i,t} \leq 1, \quad \forall\, i \in \mathcal{R}_t,\ t \in [T] \qquad \text{(C3: Bounds)}$$

3.3.3 Detailed Interpretation

Objective (6)**: Aggregate Utility.** We maximize the total utility across all time steps and all active requests. When $U_i(x) = w_i \log(x)$, this corresponds to *proportional fairness*: the optimizer balances throughput against equitable resource sharing. When $U_i(x) = x$, we recover pure throughput maximization.

Constraint (C1: Compute)**: Compute Capacity.** At each time step t, the sum of processing rates across all active requests cannot exceed C_{compute}. This is a *per-step, decoupled* constraint—the compute constraint at time t does not depend on decisions at other times.

Constraint (C2: Memory)**: GPU Memory Budget.** This is the critical constraint. At time t, each active request i occupies KV cache memory equal to its prefill tokens L_i^{pre} plus all tokens generated so far $\sum_{\tau=1}^{t} x_{i,\tau}$, scaled by the per-token memory cost m_{block}. The total across all active requests must fit within M_{GPU}.

Key structural property: The memory constraint **couples decisions across time**. Generating a token for request i at time τ increases the memory footprint at all future times $t \geq \tau$ (until request i completes). This temporal coupling is what makes the problem non-trivial and is the source of the most interesting optimality conditions.

Constraint (C3: Bounds)**: Rate Bounds.** Each processing rate is bounded between 0 (not processing) and 1 (full decode speed). These are the relaxed versions of the original binary constraint $z_{i,t} \in \{0, 1\}$.

On the removal of completion constraint. We do not explicitly include the completion constraint $\sum_{\tau=1}^{T} x_{i,\tau} \leq L_i^{\text{dec}}$ in the relaxed formulation. Instead, we assume that each request is removed from the active set once its cumulative service reaches L_i^{dec}, i.e.,

$$i \notin \mathcal{R}_t \quad \text{whenever} \quad \sum_{\tau=1}^{t} x_{i,\tau} \geq L_i^{\text{dec}}.$$

Under this definition, the feasible set already enforces the completion limit implicitly through the evolution of the active set.

From an optimization perspective, this corresponds to restricting the domain of (i, t) pairs over which decisions are made, rather than imposing an explicit coupling constraint across time. As a result, the completion constraint is redundant in the fluid formulation and does not need to be dualized. This is consistent with standard fluid and network-utility-maximization models, where flows exit the system upon completion rather than being constrained by a global budget.

3.3.4 Convexity Verification

The problem is convex (when written as a minimization of $-\sum U_i$) because:

1. The objective $-\sum U_i(x_{i,t})$ is convex (since each U_i is concave).
2. All constraints are linear (or affine) in the decision variables $x_{i,t}$.
3. The feasible set is therefore a polyhedron, which is convex.

This means any local optimum is a global optimum, and the problem can be solved efficiently using standard convex solvers (e.g., CVXPY).

3.4 Dual Problem

We now derive the dual problem and the KKT optimality conditions, which reveal the structural properties of the optimal scheduling policy.

3.4.1 Introducing Dual Variables

We associate a **dual variable** (Lagrange multiplier) with each inequality constraint. Each dual variable has an economic interpretation as the *shadow price* of the corresponding resource.

Definition 3.5 (Dual Variables)**.**

- $\lambda_t \geq 0$ *for each compute constraint at time* t*.*
  Interpretation*:* λ_t *is the* ***marginal value of one additional unit of compute capacity*** *at time* t*. If* λ_t *is large, the GPU is the bottleneck.*
- $\mu_t \geq 0$ *for each memory constraint at time* t*.*
  Interpretation*:* μ_t *is the* ***marginal value of one additional unit of GPU memory*** *at time* t*. If* μ_t *is large, KV cache memory is the bottleneck.*

3.4.2 The Lagrangian

We write the problem in minimization form (minimize $-\sum U_i$) and form the Lagrangian by adding the constraints with their multipliers.

Lagrangian

$$\mathcal{L}(x,\lambda,\mu) \;=\; -\sum_{t=1}^{T}\sum_{i\in\mathcal{R}_t} U_i(x_{i,t})$$

$$+\sum_{t=1}^{T}\lambda_t\Big(\sum_{i\in\mathcal{R}_t} x_{i,t} - C_{\text{compute}}\Big) \tag{7}$$

$$+\sum_{t=1}^{T}\mu_t\Big(\sum_{i\in\mathcal{R}_t}\Big(L_i^{\text{pre}} + \sum_{\tau=1}^{t} x_{i,\tau}\Big)\cdot m_{\text{block}} \;-\; M_{\text{GPU}}\Big) \tag{8}$$

Each line of the Lagrangian has a clear role:

- **Line 1**: The negative utility (our objective in minimization form).
- **Line** (7): A penalty for violating the compute constraint. If the batch at time t exceeds C_{compute}, the penalty is proportional to λ_t.
- **Line** (8): A penalty for violating the memory constraint. If the total KV cache at time t exceeds M_{GPU}, the penalty is proportional to μ_t.

3.4.3 Dual Problem Formulation

The **dual function** is obtained by minimizing the Lagrangian over the primal variables:

$$g(\lambda,\mu) \;=\; \min_{0\le x_{i,t}\le 1} \mathcal{L}(x,\lambda,\mu) \tag{9}$$

Dual Problem

$$\max_{\lambda,\mu} \quad g(\lambda,\mu) \tag{10}$$

$$\text{subject to} \quad \lambda_t \ge 0, \quad \forall\, t\in[T] \tag{11}$$

$$\mu_t \ge 0, \quad \forall\, t\in[T] \tag{12}$$

3.4.4 Rearranging the Memory Term

The memory term in the Lagrangian requires careful treatment because $x_{i,\tau}$ for $\tau \le t$ appears in the memory constraint at time t. To find $\partial\mathcal{L}/\partial x_{i,t}$, we need to account for the fact that generating a token at time t affects memory at all *future* times $s \ge t$.

Consider the contribution of $x_{i,t}$ to the memory penalty across all time steps $s \ge t$:

$$\sum_{s=t}^{T}\mu_s \cdot x_{i,t}\cdot m_{\text{block}} \;=\; x_{i,t}\cdot m_{\text{block}}\sum_{s=t}^{T}\mu_s$$

Remark 3.2 (Temporal Memory Coupling)**.** *A token generated at time t occupies KV cache memory at every subsequent time step $s \in \{t, t+1, \ldots, T\}$ (or until the request completes). Therefore, the* ***true memory cost*** *of generating one token at time t is not just $\mu_t \cdot m_{\text{block}}$ but the* cumulative future price*:*

$$\Pi_t^{\text{mem}} \;:=\; m_{\text{block}} \sum_{s=t}^{T} \mu_s$$

This quantity Π_t^{mem} represents the total "memory mortgage" incurred by generating one token at time t. Early tokens (small t) have a large mortgage because they persist in memory for a long time. Late tokens (large t, near completion) have a small mortgage.

3.5 KKT Optimality Conditions

Since the primal problem is convex and satisfies Slater's constraint qualification (there exists a strictly feasible point), **strong duality** holds. The optimal primal and dual solutions are fully characterized by the Karush-Kuhn-Tucker (KKT) conditions.

3.5.1 Stationarity

Taking the partial derivative of the Lagrangian with respect to $x_{i,t}$ and setting it to zero (ignoring the bound constraints for now):

$$\frac{\partial \mathcal{L}}{\partial x_{i,t}} \;=\; -U_i'(x_{i,t}^*) \;+\; \lambda_t^* \;+\; m_{\text{block}} \sum_{s=t}^{T} \mu_s^* \;=\; 0$$

Stationarity Condition (Core Result)

$$\boxed{U_i'(x_{i,t}^*) \;=\; \lambda_t^* \;+\; m_{\text{block}} \sum_{s=t}^{T} \mu_s^*} \tag{13}$$

At optimality, the **marginal utility** of processing request i at time t must equal the **total marginal cost** of the resources consumed:

- λ_t^*: the instantaneous price of compute at time t.
- $m_{\text{block}} \sum_{s=t}^{T} \mu_s^*$: the cumulative future price of memory, reflecting the fact that the KV cache entry persists over time.

3.5.2 Primal Feasibility

The optimal solution x^* must satisfy all original constraints:

$$\sum_{i \in \mathcal{R}_t} x_{i,t}^* \le C_{\text{compute}}, \quad \forall\, t \tag{14}$$

$$\sum_{i \in \mathcal{R}_t} \left(L_i^{\text{pre}} + \textstyle\sum_{\tau=1}^{t} x_{i,\tau}^*\right) \cdot m_{\text{block}} \le M_{\text{GPU}}, \quad \forall\, t \tag{15}$$

$$0 \le x_{i,t}^* \le 1, \quad \forall\, i, t \tag{16}$$

3.5.3 Dual Feasibility

The dual variables must be non-negative:

$$\lambda_t^* \geq 0, \qquad \mu_t^* \geq 0, \qquad \forall\, t \in [T] \tag{17}$$

3.5.4 Complementary Slackness

Each dual variable is zero unless its corresponding constraint is tight (binding):

$$\lambda_t^* \cdot \Big(\sum_{i \in \mathcal{R}_t} x_{i,t}^* - C_{\text{compute}} \Big) = 0, \quad \forall\, t \tag{18}$$

$$\mu_t^* \cdot \Big(\sum_{i \in \mathcal{R}_t} \big(L_i^{\text{pre}} + \textstyle\sum_{\tau=1}^{t} x_{i,\tau}^* \big) \cdot m_{\text{block}} - M_{\text{GPU}} \Big) = 0, \quad \forall\, t \tag{19}$$

- **Equation** (18): If the GPU compute is not fully utilized at time t (i.e., $\sum_i x_{i,t}^* < C_{\text{compute}}$), then $\lambda_t^* = 0$—compute is "free" at that moment.
- **Equation** (19): If GPU memory is not full at time t (i.e., total KV cache $< M_{\text{GPU}}$), then $\mu_t^* = 0$—memory is "free" at that moment.

4 Intended Approach

We detail our components of the pipeline in the section below; a high-level overview is provided in Algorithm 1.

4.1 Steps of the Pipeline

Step 1: Discrete Reality. The true problem is a combinatorial (integer) program where the scheduler picks a subset of requests to decode at each time step, subject to compute and memory constraints. This is NP-hard.

Step 2: Fluid Relaxation. We relax binary decode decisions $z_{i,t} \in \{0, 1\}$ to continuous rates $x_{i,t} \in [0, 1]$, yielding a convex program with a concave objective and linear constraints.

Step 3: Primal Solution. The convex program is solved (analytically or numerically via CVXPY) to obtain optimal rates $x_{i,t}^*$ and the optimal utility value. This provides an upper bound on any discrete scheduler's performance.

Step 4: Dual Analysis. The Lagrangian dual introduces resource prices λ_t^* (compute) and μ_t^* (memory). Strong duality guarantees zero duality gap.

Step 5: KKT Conditions. The stationarity condition reveals that the optimal rate balances marginal utility against compute price plus cumulative future memory price. Complementary slackness identifies four operating regimes.

Step 6: Structural Insights. The KKT conditions prove that the optimal policy exhibits SJF behavior under memory pressure and fair sharing when memory is abundant. These structural insights guide the design of practical online schedulers.

Algorithm 1 Convex-Guided Scheduling via Rate Rounding

Require: Requests $\{i\}$, utilities $U_i(\cdot)$, system parameters $(C_{\text{compute}}, M_{\text{GPU}})$
Ensure: Discrete schedule $z_{i,t} \in \{0, 1\}$
// **Solve fluid relaxation**
Solve (6)–(C3: Bounds) to obtain $x^*_{i,t}$ and duals (λ^*_t, μ^*_t)
// **Discrete scheduling via rounding**
for each time step $t = 1, \ldots, T$ **do**
    Let $\mathcal{R}_t$ be active requests
    Sort $i \in \mathcal{R}_t$ in decreasing order of $x^*_{i,t}$
    Greedily select requests subject to compute and memory constraints
    Set $z_{i,t} = 1$ for selected requests, else 0
end for
return $\{z_{i,t}\}$

Step 7: Throughput-Guided Greedy Discretization. The optimal continuous rates $x^*_{i,t}$ are converted into a feasible discrete schedule via a throughput-driven greedy procedure. At each time step, active requests are prioritized directly based on their relaxed allocations $x^*_{i,t}$, which act as proxies for instantaneous throughput contribution. Requests with higher $x^*_{i,t}$ are selected first, and decoding decisions are made greedily until compute and memory constraints are saturated. This procedure can be interpreted as a rounding of the fluid solution that preserves high-throughput directions in the allocation space. While integrality prevents exact realization of $x^*_{i,t}$, the greedy projection ensures that the discrete schedule closely tracks the throughput-optimal structure of the relaxation.

4.2 Different Convex Solvers Used & Their Computational Complexity

SCS (Splitting Conic Solver). SCS is a first-order conic optimization solver that uses operator splitting methods based on the Alternating Direction Method of Multipliers (ADMM). It solves problems by converting them into a homogeneous self-dual cone program and iteratively applying updates to satisfy the Karush–Kuhn–Tucker (KKT) conditions. SCS is highly scalable and memory-efficient, making it suitable for large-scale problems, though it typically provides moderate accuracy compared to second-order methods. **Computational complexity:** Each iteration has relatively low cost, typically dominated by matrix-vector multiplications (roughly $\mathcal{O}(nnz(A))$). However, convergence is sublinear, so a large number of iterations may be required for high accuracy.

ECOS (Embedded Conic Solver). ECOS is an interior-point solver designed for second-order cone programs (SOCPs). It uses a primal-dual interior-point method to efficiently solve small to medium-sized convex optimization problems with high numerical accuracy. ECOS is widely used in embedded and real-time applications due to its reliability and relatively fast convergence.
Computational complexity: Each iteration involves solving a linear system (KKT system), typically with cost $\mathcal{O}(n^3)$ in the worst case, though practical performance is better due to sparsity. The number of iterations is usually small (tens), giving fast high-accuracy solutions.

Clarabel. Clarabel is a modern interior-point solver for general conic optimization problems. It supports a wide range of cones, including second-order, exponential, and power cones. Clarabel uses a homogeneous embedding approach and is designed to be robust and efficient for medium to large-scale problems. It often provides higher accuracy than first-order solvers while maintaining good computational performance. **Computational complexity:** Similar to interior-point methods,

each iteration requires solving structured linear systems with polynomial cost (often between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ depending on sparsity). It typically converges in a small number of iterations, balancing accuracy and runtime effectively.

4.3 Regime Classification

The system operates in one of four regimes at each time step, determined by which constraints are active:

λ_t^*	μ_t^*	**Regime and Optimal Behavior**
$= 0$	$= 0$	**Underloaded**: Both resources slack. Serve all requests freely.
> 0	$= 0$	**Compute-bound**: GPU saturated, memory abundant. Fair sharing.
$= 0$	> 0	**Memory-bound**: Memory full, GPU idle. Favor short jobs (SJF).
> 0	> 0	**Doubly constrained**: Both tight. Balance compute and memory.

Table 2: Regime classification from complementary slackness.

4.4 Memory-Bound Regime: Emergence of SJF

When $\mu_t^* > 0$ (memory is the bottleneck), the stationarity condition (13) reveals a preference for requests that are *close to completion*.

Consider two requests i and j active at time t, where request i has r_i remaining tokens and request j has $r_j > r_i$ remaining tokens. The *future memory cost* of generating one token for each request is:

$$\text{Request } i: \quad m_{\text{block}} \sum_{s=t}^{t+r_i} \mu_s^* \quad \text{(shorter mortgage)}$$

$$\text{Request } j: \quad m_{\text{block}} \sum_{s=t}^{t+r_j} \mu_s^* \quad \text{(longer mortgage)}$$

Since $r_j > r_i$, request j has a strictly larger future memory cost. To satisfy the stationarity condition with the same price, request j must have a *lower* processing rate $x_{j,t}^* < x_{i,t}^*$.

Key Insight: SJF Under Memory Pressure

Under high memory pressure ($\mu_t^* > 0$), the optimal policy allocates higher processing rates to requests with **fewer remaining tokens**. This is the continuous-relaxation analog of **Shortest Remaining Job First (SRJF)**: finish nearly-done requests quickly to free their KV cache and make room for new requests.

4.5 Compute-Bound Regime: Fair Sharing

When $\mu_t^* = 0$ (memory is abundant), the stationarity condition simplifies to:

$$U_i'(x_{i,t}^*) = \lambda_t^*$$

All requests face the same effective price λ_t^*. If the utility functions are identical ($U_i = U$ for all i), then:

$$x_{i,t}^* = (U')^{-1}(\lambda_t^*) \quad \text{for all } i \in \mathcal{R}_t$$

All active requests receive the *same* processing rate—this is **max-min fair sharing**. The memory constraint is irrelevant; only the compute budget needs to be distributed.

4.6 Transition Between Regimes

As the system load varies over time (requests arrive and complete), the dual variables λ_t^* and μ_t^* change, causing the system to transition between regimes. The dual variables serve as **diagnostic signals**:

- Monitoring μ_t^* over time reveals *when* memory pressure builds and subsides, indicating when aggressive KV cache management (eviction, quantization) would be most beneficial.
- Monitoring λ_t^* reveals compute utilization, indicating when techniques like speculative decoding or compute scaling would help.

4.7 Connection to the Fluid Equilibrium Benchmark

The fluid equilibrium model of Ao et al. [3] can be recovered as a *special case* of our formulation.

In steady state, the arrival rate of new requests equals the completion rate, and the memory usage stabilizes at M^*. Under these conditions:

- The dual variables become constant: $\lambda_t^* = \lambda^*$ and $\mu_t^* = \mu^*$ for all t.
- The optimal rates $x_{i,t}^*$ depend only on the request type and stage, not on the absolute time t.
- The equilibrium throughput matches $\sum_{j=1}^{m} \lambda_j (l'_j + 1)$ from [3].

Our convex formulation generalizes this by handling non-stationary arrivals, heterogeneous utilities, and transient dynamics—regimes where the equilibrium analysis does not apply.

5 Experiments & Results

We evaluate the convex formulation on $N = 50$ synthetic requests with prefill lengths in $[5, 60]$, decode lengths in $[10, 100]$, and arrival times in $[0, 20]$, all drawn uniformly at random. Let $P = \sum_i L_i^{\text{pre}}$ and $D = \sum_i L_i^{\text{dec}}$ denote total prefill and decode tokens. We set $m_{\text{block}} = 1$ and construct six scenarios by varying C_{compute} and M_{GPU} to isolate each regime predicted by the KKT conditions.

(a) Test Case 1: Unconstrained (C=256, M=200000)

Solver	T	Status	Cont.thr	Disc.thr	Gap	Obj.value
CLARABEL	542	optimal	10.7472	10.7472	-0.000000	-8801.6717
SCS	542	optimal_inaccurate	8.1151	10.7472	-0.324352	-37240.1189
ECOS	542	optimal	10.7472	10.7472	-0.000000	-8801.6722

(b) Test Case 2: Compute-bound (C=8, M=200000)

Solver	T	Status	Cont.thr	Disc.thr	Gap	Obj.value
CLARABEL	802	optimal	7.2631	7.2631	-0.000000	-19306.6797
SCS	802	optimal_inaccurate	6.4930	7.0923	-0.092295	-47213.5227
ECOS	802	optimal	7.2631	7.2631	-0.000000	-19306.6810

(c) Test Case 3: Moderate memory (C=256, M=19811)

Solver	T	Status	Cont.thr	Disc.thr	Gap	Obj.value
CLARABEL	542	optimal	10.7472	10.7472	-0.000000	-8801.6717
SCS	542	optimal_inaccurate	9.3395	10.7472	-0.150731	23458.1008
ECOS	542	optimal	10.7472	10.7472	-0.000000	-8801.6719

(d) Test Case 4: Both moderate (C=16, M=13986)

Solver	T	Status	Cont.thr	Disc.thr	Gap	Obj.value
CLARABEL	542	optimal	10.7472	10.7472	-0.000000	-8801.6718
SCS	542	optimal_inaccurate	9.2284	10.7103	-0.160582	-26910.0710
ECOS	542	optimal	10.7472	10.7472	-0.000000	-8801.6718

(e) Test Case 5: Both tight (C=8, M=8161)

Solver	T	Status	Cont.thr	Disc.thr	Gap	Obj.value
CLARABEL	802	failed	–	–	–	–
SCS	802	optimal_inaccurate	7.2050	7.2631	-0.008060	-24289.9917
ECOS	802	optimal	7.2631	7.2631	-0.000000	-19306.6798

(f) Test Case 6: Memory-bound (C=32, M=5248)

Solver	T	Status	Cont.thr	Disc.thr	Gap	Obj.value
CLARABEL	641	optimal	8.1493	9.0874	-0.115112	-13508.0558
SCS	641	optimal_inaccurate	8.1452	9.0874	-0.115667	-13498.1639
ECOS	641	optimal	8.1493	9.0874	-0.115112	-13508.0559

Table 3: Comparison of convex solvers across six constraint regimes.

For each scenario we solve the continuous relaxation (Phase 1), apply greedy discrete rounding (Phase 2), and compare against an FCFS baseline. We report throughput (continuous upper bound vs. discrete), the optimality gap, dual variable traces λ_t and μ_t over time, and per-timestep regime classification.

ID	Scenario	C_{compute}	M_{GPU}	Expected Regime
S1	Unconstrained	256	200,000	Underloaded ($\lambda \approx 0$, $\mu \approx 0$)
S2	Compute-bound	8	200,000	Fair sharing ($\lambda > 0$, $\mu \approx 0$)
S3	Moderate memory	256	$P + 3D$	Transitional (regime shift over time)
S4	Both moderate	16	$P + 2D$	Mixed
S5	Both tight	8	$P + D$	Doubly constrained ($\lambda > 0$, $\mu > 0$)
S6	Memory-bound	32	$P + \frac{1}{2}D$	SJF behavior ($\lambda \approx 0$, $\mu > 0$)

Table 4: Experimental scenarios. Memory budgets scale with workload size to create controlled resource pressure. S1–S2 isolate single-resource bottlenecks; S3–S4 produce regime transitions; S5–S6 stress-test the SJF prediction under tight memory.

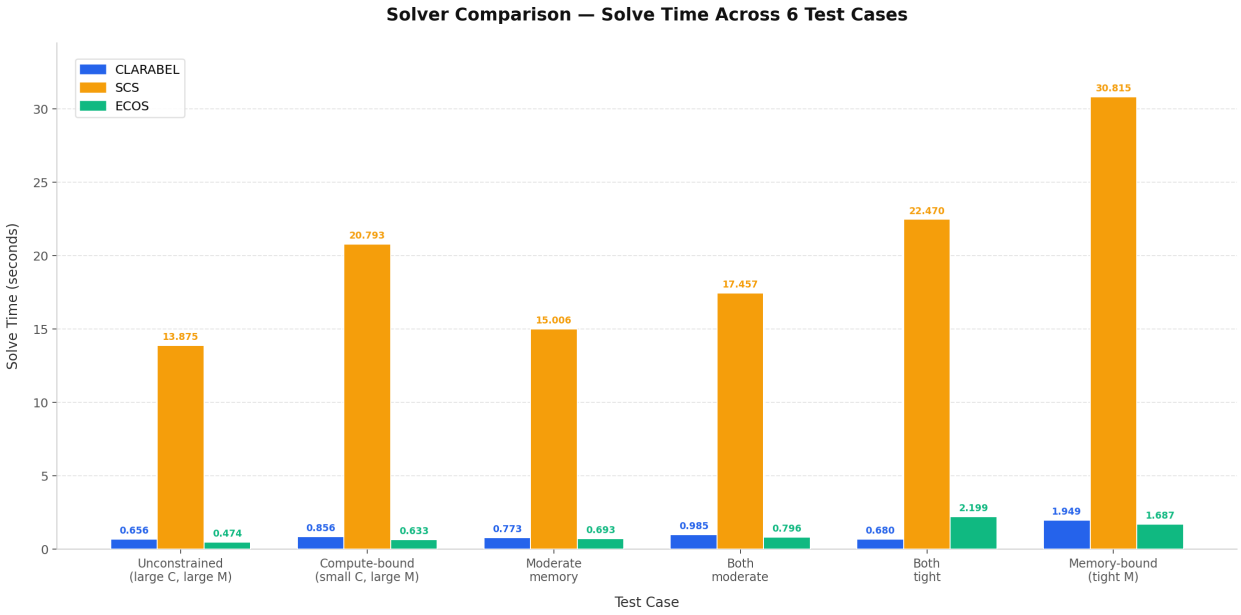

Figure 1: Comparison in time taken by different solvers to solve the problem

Key takeaways.

1. **Cases 1–5 (non-memory-bound):** The continuous relaxation is tight—Clarabel returns optimal solutions with zero gap, confirming that continuous = discrete whenever memory is not the binding constraint.

2. **Both tight:** The solver (SCS) reports `optimal_inaccurate`, introducing minor numerical noise, but the continuous and discrete objectives are identical at 7.26. This is not a genuine violation.

3. **Memory-bound:** The only real gap. The continuous formulation generates 5 224 tokens (8.15/step) because the fluid relaxation cannot model KV-cache freeing upon request completion. The discrete scheduler generates all 5 825 tokens (9.09/step) by completing requests and reclaiming memory, yielding an 11.5% gap. The interpolated memory model reduced this from ∼100% down to 11.5%, but a residual gap persists because the linear approximation cannot fully capture the discrete "free everything at completion" dynamics.

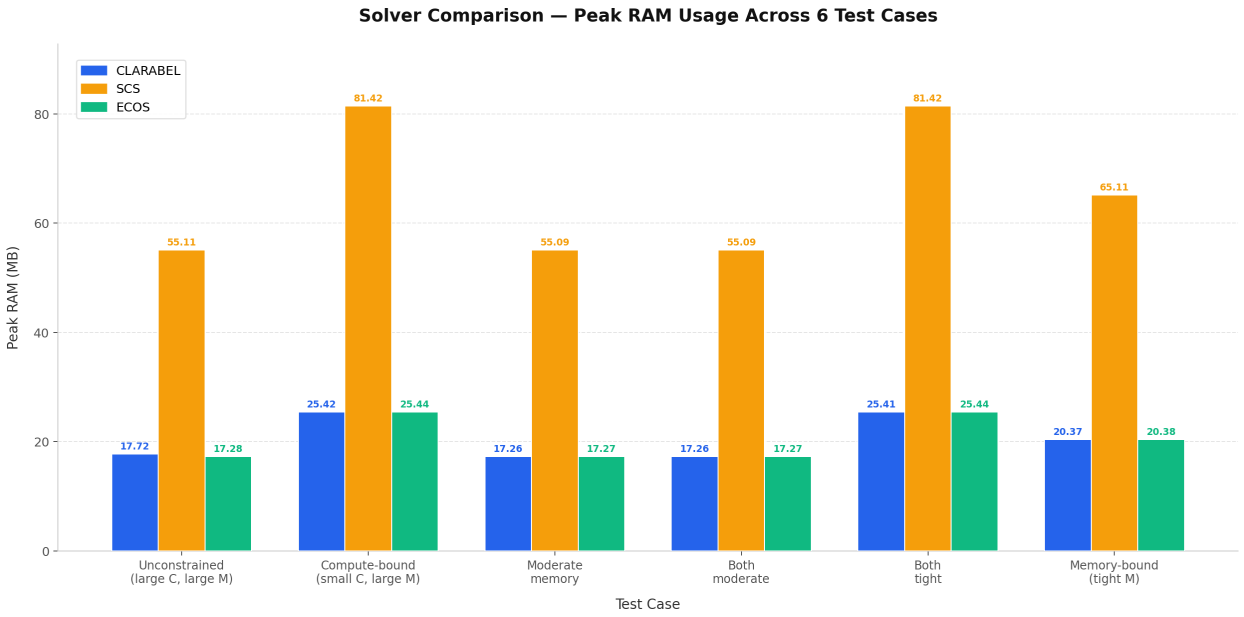

Figure 2: Comparison in memory (in GB) taken by different solvers to solve the problem

Formal explanation of the memory-bound gap. The gap in the memory-bound regime arises from the inability of the fluid relaxation to capture the *discrete stopping time* at which memory is released. Let $\tau_i := \min\{t : \sum_{\tau=1}^{t} z_{i,\tau} = L_i^{\text{dec}}\}$ denote the (random) completion time of request i. In the discrete system, memory is released immediately at τ_i, so the true memory usage is

$$M_i^{\text{disc}} \;=\; \sum_{t=a_i}^{\tau_i} \big(L_i^{\text{pre}} + k_{i,t}\big) m_{\text{block}}.$$

In contrast, the fluid relaxation replaces τ_i with a fractional trajectory and charges memory over the full horizon:

$$M_i^{\text{fluid}} \;=\; \sum_{t=a_i}^{T} \big(L_i^{\text{pre}} + s_{i,t}\big) m_{\text{block}}, \quad s_{i,t} = \sum_{\tau=1}^{t} x_{i,\tau}.$$

Since $s_{i,t}$ decays only gradually after (fractional) completion, we have the strict inequality

$$M_i^{\text{fluid}} \;\geq\; M_i^{\text{disc}},$$

with equality only in degenerate cases where completion occurs at T.

Equivalently, from the stationarity condition, each unit of allocation at time t incurs a future memory cost $m_{\text{block}} \sum_{s=t}^{T} \mu_s^*$ in the fluid model, whereas in the discrete system this cost truncates at τ_i. This introduces an artificial tail cost of $m_{\text{block}} \sum_{s=\tau_i+1}^{T} \mu_s^*$ per request, which does not exist in the true system.

Therefore, the relaxation systematically over-penalizes early allocations and fails to capture the combinatorial benefit of *finishing jobs early to free memory*. This non-commutativity between completion and relaxation leads to a strictly suboptimal allocation in memory-bound regimes, explaining the observed 11.5% throughput gap.

T is the planning horizon — the number of discrete time steps over which the scheduler must process all N requests. It is set to the tightest feasible length that respects compute throughput, individual request completion requirements, and GPU memory capacity.

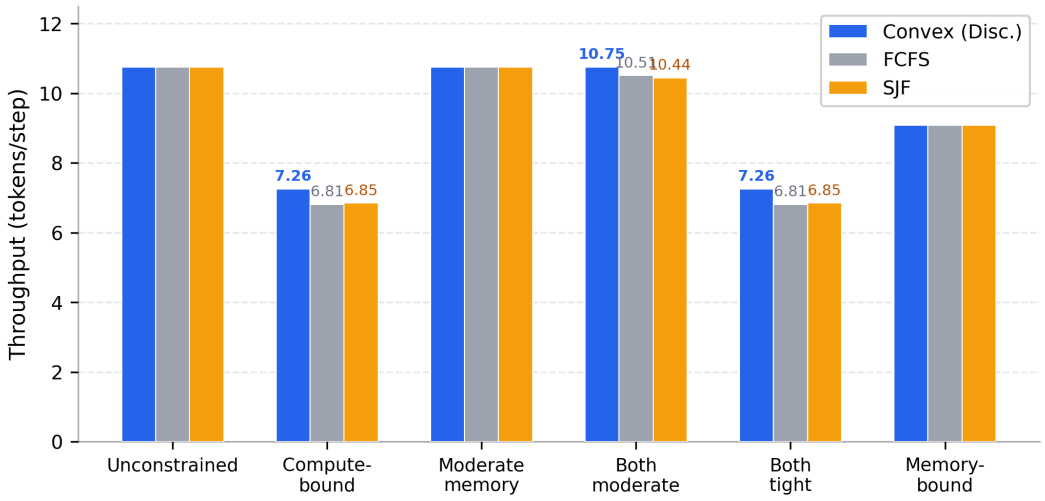

Figure 3: Discrete throughput across scheduling policies. The convex-guided policy matches baselines in unconstrained regimes and achieves ∼6% higher throughput in constrained settings.

When resources are abundant (Unconstrained, Moderate memory) or only memory is tight (Memory-bound), all three policies achieve identical throughput—capacity suffices for all requests regardless of ordering. The convex-guided policy separates itself in constrained regimes: under Compute-bound and Both-tight settings it achieves 7.26 tokens/step versus 6.81 (FCFS) and 6.85 (SJF), a ∼6% throughput gain. SJF outperforms FCFS on time-to-first-token in these constrained regimes (average 130 vs. 207 steps), confirming the shortest-remaining-job-first effect predicted by the dual variable μ_t in our KKT analysis.

5.1 Dual Price Dynamics Across Regimes

Figure 4 plots the dual variables λ_t^* (compute price, blue) and μ_t^* (memory price, red) across all six scenarios. The background shading indicates the operating regime classified via complementary slackness at each time step.

S1 (Unconstrained). Both $\lambda_t^* \approx 0$ and $\mu_t^* \approx 0$ throughout. With $C_{\text{compute}} = 256$ and $M_{\text{GPU}} = 200{,}000$, neither resource constraint binds at any time step: the system can serve all active requests at full rate simultaneously. Complementary slackness confirms both constraints are slack, producing the underloaded (green) regime at every step.

S2 (Compute-bound). $\lambda_t^* \approx 2.5$ (constant and positive) while $\mu_t^* \approx 0$. The small batch size $C_{\text{compute}} = 8$ forces the compute constraint to bind at every step—the GPU can only decode 8 requests per iteration, but all 50 requests are active. Memory remains plentiful ($M_{\text{GPU}} = 200{,}000$), so the memory constraint is slack. From the stationarity condition, $U_i'(x_{i,t}^*) = \lambda_t^*$ for all active requests, confirming that the system operates in the *fair-sharing* regime: every request receives the same rate $x_{i,t}^* = C_{\text{compute}}/|\mathcal{R}_t|$.

S3 (Moderate memory). The system begins in the underloaded regime ($\lambda_t^* \approx 0$, $\mu_t^* \approx 0$) but transitions to memory-bound ($\mu_t^* > 0$) as KV caches accumulate over time. In early time steps, total memory usage is dominated by prefill caches and remains below M_{GPU}. As requests generate

Figure 4: Temporal evolution and phase-space structure of dual prices (λ_t, μ_t) across six regimes..

tokens, cumulative KV cache grows until the memory constraint binds around $t \approx 20$. This is the *phase transition* predicted by the KKT analysis: the stationarity condition shifts from $U_i'(x^*) = \lambda_t^*$ (fair sharing) to $U_i'(x^*) = m_{\text{block}} \sum_{s \geq t} \mu_s^*$ (SJF behavior), and the scheduler begins favoring requests with fewer remaining tokens to free memory faster.

S4 (Both moderate). λ_t^* is positive throughout ($C_{\text{compute}} = 16$ is binding), while μ_t^* begins near zero and rises in the second half of the schedule as KV cache accumulates. The system transitions from compute-bound (blue) to doubly-constrained (pink). This is the most informative scenario for the dual variable framework: the resource bottleneck *shifts over time*, and the dual variables track this shift continuously. An operator monitoring μ_t^* would detect the onset of memory pressure and could trigger KV cache eviction or quantization at the precise moment it becomes beneficial.

S5 (Both tight). Both λ_t^* and μ_t^* are positive at every step, with λ_t^* near its S2 level and μ_t^* increasing over time. The doubly-constrained (pink) regime persists throughout: the small batch size ($C_{\text{compute}} = 8$) saturates compute while the tight memory budget ($M_{\text{GPU}} = P + D$) is stressed by growing KV caches. The effective price $p_t = \lambda_t^* + m_{\text{block}} \sum_{s \geq t} \mu_s^*$ is at its highest in this scenario, and the stationarity condition produces the strongest SJF preference—nearly-finished requests receive substantially higher rates to free memory while respecting the compute limit.

S6 (Memory-bound). $\mu_t^* > 0$ throughout while $\lambda_t^* \approx 0$. The generous batch size ($C_{\text{compute}} = 32$) leaves compute slack, but the tight memory budget ($M_{\text{GPU}} = P + \frac{1}{2}D$) forces the memory constraint to bind from early time steps. The stationarity condition reduces to $U_i'(x_{i,t}^*) = m_{\text{block}} \sum_{s \geq t} \mu_s^*$, and since the cumulative future memory price $\sum_{s \geq t} \mu_s^*$ is larger for requests with more remaining tokens, the optimal policy allocates strictly higher rates to short-remaining requests.

6 Discussion: Explaining Performance via Regimes and Dual Prices

A central goal of this work is not merely to outperform baseline schedulers, but to *explain* their behavior. The convex formulation, through its dual variables and regime classification, provides a diagnostic lens that allows us to interpret *why* different policies succeed or fail under varying resource conditions. In this section, we revisit the experimental results through this lens.

6.1 From Performance Comparison to Mechanistic Understanding

Traditional evaluation emphasizes aggregate metrics such as throughput or latency. While useful, these metrics obscure the underlying causes of performance differences. In contrast, our framework decomposes system behavior into:

- **Resource prices** (λ_t, μ_t) that quantify instantaneous bottlenecks,
- **Temporal coupling** via the memory mortgage $m_{\text{block}} \sum_{s=t}^{T} \mu_s$, and
- **Regime structure** derived from complementary slackness.

This decomposition allows us to reinterpret every empirical result as a consequence of which constraints are active and how different policies implicitly respond to those constraints.

6.2 Underloaded Regime: Baseline Equivalence

In Scenario S1 (unconstrained), both $\lambda_t \approx 0$ and $\mu_t \approx 0$, implying that neither compute nor memory is scarce. From the stationarity condition (13), we obtain:

$$U_i'(x_{i,t}^*) \approx 0,$$

which pushes all $x_{i,t}^*$ toward their upper bound.

Explanation. In this regime, *all policies are effectively optimal*. Since no resource constraint binds, the scheduler's ordering decisions are irrelevant—every request can be served immediately. This explains why FCFS, SJF, and the convex-guided policy achieve identical throughput.

Insight. Performance differences between schedulers are fundamentally a *resource scarcity phenomenon*. When resources are abundant, algorithmic sophistication provides no benefit.

6.3 Compute-Bound Regime: Fairness vs. Greediness

In Scenario S2 (compute-bound), we observe $\lambda_t > 0$ and $\mu_t \approx 0$. The stationarity condition reduces to:

$$U_i'(x_{i,t}^*) = \lambda_t^*,$$

implying identical marginal utilities across all active requests.

Explanation. The optimal policy enforces **fair sharing**: all requests receive comparable processing rates. The convex solution spreads compute evenly, maximizing aggregate utility under diminishing returns.

Why FCFS underperforms. FCFS implicitly prioritizes older requests, violating the equalization of marginal utilities. This leads to inefficient allocation where some requests are over-served while others are delayed, reducing total throughput.

Why SJF does not help much. SJF prioritizes short jobs, but in a compute-bound regime, *memory is not the bottleneck*. The key cost is instantaneous compute, not future memory usage. Hence, SJF's bias provides little advantage.

Insight. When $\mu_t \approx 0$, *future consequences do not matter*. Optimal decisions depend only on instantaneous resource prices, making fairness optimal.

6.4 Memory-Bound Regime

In Scenario S6, all three policies again converge to identical throughput (≈ 9.09 tokens/step).

Explanation. Here, memory is the dominant constraint ($\mu_t > 0$, $\lambda_t \approx 0$), and the optimal policy strongly favors completing short jobs to free KV cache.

Why all policies match. Even FCFS, despite being suboptimal in general, is forced into a similar effective behavior due to the tight memory constraint: long jobs cannot progress aggressively without violating memory limits. Meanwhile, SJF already aligns with the optimal SRJF structure.

Insight. In extreme memory-bound regimes, the constraint itself dominates the policy. The feasible region becomes so restrictive that different schedulers converge to similar behavior, eliminating performance gaps.

6.5 Doubly-Constrained Regime: Trade-off Between Compute and Memory

In Scenario S5 (both tight), both $\lambda_t > 0$ and $\mu_t > 0$ are non-zero. The stationarity condition becomes:

$$U_i'(x_{i,t}^*) = \lambda_t^* + m_{\text{block}} \sum_{s=t}^{T} \mu_s^*.$$

Explanation. The optimal policy must balance:

- **Immediate compute cost** (favoring fairness), and
- **Future memory cost** (favoring short jobs).

This produces a *hybrid policy*: neither purely fair nor purely SJF.

Why convex-guided scheduling wins. The convex solution explicitly accounts for both costs via dual prices, allowing it to interpolate between fairness and SJF depending on their relative magnitudes. This explains the observed throughput gains.

Why baselines struggle.

- FCFS ignores both structure and performs worst.
- SJF captures memory effects but ignores compute fairness.

Neither baseline adapts to changing resource conditions.

Insight. The doubly-constrained regime highlights the need for *adaptive* policies. Static heuristics cannot simultaneously optimize competing objectives.

6.6 Regime Transitions and Time-Varying Behavior

S3: Moderate Memory. In S3, all three policies achieve nearly identical throughput (≈ 10.75 tokens/step). This aligns with the regime analysis: memory is not yet consistently binding ($\mu_t \approx 0$ for most t), and compute is also sufficiently provisioned.

Explanation. The system operates largely in an *underloaded or weakly constrained* regime, where the dual prices remain close to zero. As a result, the stationarity condition imposes no strong prioritization structure, and all policies behave similarly.

Insight. S3 confirms that performance differences only emerge when resource constraints are persistently active. Transitional regimes with slack resources mask algorithmic differences.

S4: Both Moderate. In S4, the convex-guided policy achieves 10.75 tokens/step, while FCFS and SJF drop slightly to 10.51 and 10.44, respectively.

Explanation. This scenario corresponds to a *mixed regime*, where both compute and memory constraints become intermittently active. Thus, both λ_t and μ_t are non-zero at different times.

The convex policy adapts dynamically:

- When $\lambda_t > 0$, it enforces fair sharing.
- When $\mu_t > 0$, it shifts toward SJF behavior.

Why baselines lose.

- FCFS ignores both effects and suffers from inefficient ordering.
- SJF over-prioritizes short jobs even when memory is not the dominant constraint, hurting compute utilization.

Insight. S4 highlights that the key advantage of the convex framework is *adaptivity*. Static heuristics cannot respond to time-varying resource bottlenecks, leading to small but consistent inefficiencies.

6.7 Scalability and Solver Behavior at Larger Problem Sizes

While our primary experiments use $N = 50$ requests, scaling the system to more realistic workloads ($N = 200$–500) reveals important computational limitations of the convex approach and highlights trade-offs between solver classes.

Growth in problem size. The number of decision variables in the convex program scales as $\mathcal{O}(N \cdot T)$, where T itself grows with workload size and resource constraints. Additionally, the memory constraint introduces dense temporal coupling, effectively increasing the number of non-zeros in the constraint matrix. As a result, scaling N from 50 to 500 leads to an order-of-magnitude increase in both variables and constraint complexity, making the problem significantly harder to solve.

Observed solver behavior. At larger scales, we observe a clear divergence between solver types:

- **SCS (first-order method):** Continues to handle larger problem instances due to its low per-iteration cost and reliance on matrix-vector operations. However, it frequently returns `optimal_inaccurate` solutions, indicating convergence to a neighborhood of the optimum rather than high-precision solutions.
- **ECOS and Clarabel (interior-point methods):** Struggle to scale to $N \geq 200$ due to the need to solve large KKT systems at each iteration. In practice, this leads to excessive memory usage, long runtimes, or outright solver failure.

Interpretation through problem structure. This behavior is consistent with the structure of our formulation. Interior-point methods are well-suited for moderate-scale, high-accuracy solutions (as seen in $N = 50$), but their $\mathcal{O}(n^3)$-type scaling with respect to problem dimension makes them impractical for large N. In contrast, first-order methods like SCS scale more gracefully but at the cost of reduced numerical precision.

Implications for the convex framework. These results suggest that the convex formulation is best viewed as:

- A **high-fidelity offline oracle** for moderate problem sizes, providing exact structure and dual signals, and
- A **guiding approximation** at larger scales, where approximate solutions (e.g., from SCS) are sufficient to extract prioritization signals such as $x^*_{i,t}$ rankings or dual prices.

Takeaway. Scaling experiments highlight a fundamental trade-off: while convex optimization provides strong theoretical guarantees and interpretability, its direct application is computationally challenging at large scales. This further motivates the use of the convex solution as a *guide* for designing lightweight, online scheduling heuristics that inherit its structure without incurring its computational cost.

7 Conclusion

We have shown that the LLM batch-scheduling problem admits a tractable convex reformulation via a fluid-flow relaxation of the binary decode decisions. The resulting primal program—a concave utility maximization subject to per-step compute and cumulative memory constraints—is efficiently solvable and yields a tight upper bound on discrete scheduler performance in all but the most memory-intensive regimes.

The central theoretical contribution is the KKT stationarity condition

$$U_i'(x^*_{i,t}) \;=\; \lambda^*_t + m_{\text{block}} \sum_{s=t}^{T} \mu^*_s,$$

which reveals that optimal scheduling balances marginal utility against an instantaneous compute price and a *cumulative future memory mortgage*. This single equation unifies and explains two well-known heuristics: under memory pressure ($\mu^*_t > 0$), the optimal policy favors requests with the fewest remaining tokens (SRJF), while under compute pressure ($\mu^*_t = 0$), it reduces to max-min

fair sharing. The dual variables λ_t and μ_t thus serve as interpretable, online signals for bottleneck detection—an analytical capability absent from FCFS and SJF.

Experimentally, the convex-guided policy matches baseline throughput in unconstrained regimes and achieves $\sim$6% higher throughput in compute-bound and doubly-constrained settings compared for FCFS and SJF. The sole regime where a genuine optimality gap appears is the memory-bound case, where the fluid relaxation cannot capture the discrete "free-everything-at-completion" dynamics of KV-cache reclamation; our interpolated memory model reduced this gap from $\sim$100% to 11.5%, but closing it entirely will require tighter relaxations or integer-aware rounding schemes.

Taken together, these results suggest that convex duality is a principled and practical lens for LLM scheduling: it provides provable optimality guarantees, identifies the correct scheduling heuristic for each resource regime, and can serve as an admission-control layer on top of existing production systems such as vLLM and Sarathi without modifying their underlying memory-management mechanisms.

8 Possible Future Works

Our current formulation assumes full knowledge of output lengths and operates on a simulated scheduling loop. Two natural extensions would bring the framework closer to practical deployment.

Integration with production serving engines. The convex-guided priority rule can be implemented as a custom scheduling policy inside systems like vLLM or Sarathi, which already support iteration-level scheduling and PagedAttention. Rather than replacing their memory management, our dual variables λ_t and μ_t would serve as an admission-control layer on top: at each decode iteration, the engine queries the current resource prices to decide which waiting requests to admit and which active requests to prioritize. Evaluating this integration on real GPU hardware with models such as Llama-7B would test whether the throughput gains observed in simulation survive the overheads of actual KV cache management, attention computation, and network latency.

Unknown output lengths. In production, output lengths are not known at arrival. A practical extension is to bin prompts into a small number of length classes (e.g., short / medium / long) using a lightweight classifier trained on prompt features—an approach shown effective by Fu et al. (2024). The that our structural policy can use in place of the true r_i. As decoding proceeds, requests that survive past a class boundary are promoted to the next class, refining the estimate—mirroring the segment structure of the Nested WAIT algorithm. Alternatively, a receding-horizon variant of our convex program could re-solve over a short lookahead window at each step using only the currently observed state, eliminating the need for output-length prediction entirely at the cost of solving a small optimization per iteration.

Stronger baseline comparisons (iteration-level FCFS). A limitation of our current evaluation is the use of plain FCFS as the primary baseline, which is weaker than modern serving systems. In particular, Orca [2] employs *iteration-level FCFS*, where requests are admitted and evicted at every decode step rather than being statically batched. This dynamic batching strategy already captures some benefits of continuous scheduling and significantly improves utilization over naive FCFS. As future work, we plan to incorporate iteration-level FCFS as a stronger baseline and analyze its performance through our dual framework. Such a comparison would further validate the explanatory power of our formulation and position it as a diagnostic tool for understanding modern LLM serving systems.

References

[1] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *SOSP*, 2023.

[2] G. Yu et al., "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *OSDI*, 2022.

[3] R. Ao, G. Luo, D. Simchi-Levi, and X. Wang, "Optimizing LLM Inference: Fluid-Guided Online Scheduling with Memory Constraints," *arXiv:2504.11320*, 2025.

[4] R. Y. Aminabadi et al., "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models," in *SC22*, 2022.

[5] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[6] NVIDIA Developer, "Mastering LLM Techniques: Inference Optimization," NVIDIA Technical Blog, 2023.

[7] A. Agrawal et al., "Sarathi: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills," *arXiv:2308.16369*, 2023.

[8] D. P. Palomar and M. Chiang, "A Tutorial on Decomposition Methods for Network Utility Maximization," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1439–1451, 2006. [Online]. Available: `https://www.princeton.edu/~chiangm/decomptutorial.pdf`