MEETING SLOS, SLASHING HOURS: AUTOMATED ENTERPRISE LLM OPTIMIZATION WITH OPTIKIT

Nicholas Santavas [1 2] **Kareem Eissa** [1] **Patrycja Cieplicka** [1] **Piotr Florek** [1] **Matteo Nulli** [1] **Stefan Vasilev** [1]
Seyyed Hadi Hashemi [1] **Antonios Gasteratos** [2] **Shahram Khadivi** [1]

ABSTRACT

Enterprise LLM deployment faces a critical scalability challenge: organizations must optimize models systematically to scale AI initiatives within constrained compute budgets, yet the specialized expertise required for manual optimization remains a niche and scarce skillset. This challenge is particularly evident in managing GPU utilization across heterogeneous infrastructure while enabling teams with diverse workloads and limited LLM optimization experience to deploy models efficiently. We present OPTIKIT, a distributed LLM optimization framework that democratizes model compression and tuning by automating complex optimization workflows for non-expert teams. OPTIKIT provides dynamic resource allocation, staged pipeline execution with automatic cleanup, and seamless enterprise integration. In production, it delivers more than 2× GPU throughput improvement while empowering application teams to achieve consistent performance improvements without deep LLM optimization expertise. We share both the platform design and key engineering insights into resource management, pipeline orchestration, and integration patterns that enable large-scale, production-grade democratization of model optimization. Finally, we open-source the system to enable external contributions and broader reproducibility.

1 INTRODUCTION

The proliferation of Large Language Models (LLMs) (Brown et al., 2020; Aaron Grattafiori, 2024; An Yang, 2025) across enterprises has created a major computational challenge (Chavan et al., 2024). As organizations adopt generative AI, they face a fundamental tension between the exponential growth in demand for AI-driven features and the finite and expensive supply of specialized GPU infrastructure. This scalability issue, if unaddressed, threatens to stifle innovation and render the widespread deployment of powerful LLMs economically untenable. At global technology companies, like eBay, this is not a distant prospect but an immediate operational reality. The ambition to enhance user experience with a new generation of LLM-powered applications is constrained by hardware capacity and operational efficiency. Deploying models from 8B to over 70B parameters creates a significant strain on computational resources. Manual optimization (Zhu et al., 2024), a specialized craft practiced by few experts, does not scale, while existing tools often lack the robustness and seamless integration required for production systems (Park et al., 2025). This gap forces a trade-off between feature velocity and performance, creating dependencies on a small pool of experts.

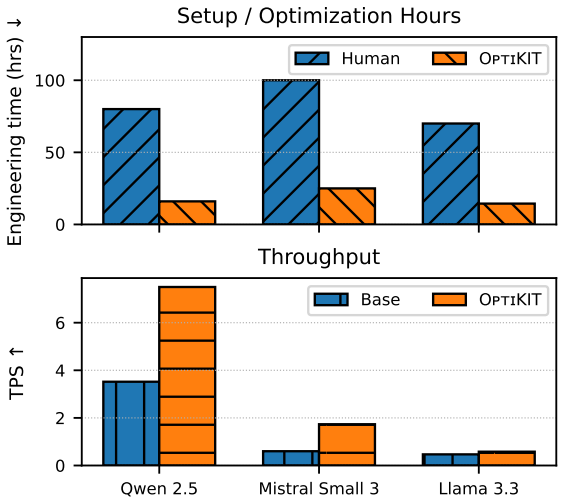


Figure 1: **OPTIKIT time and throughput gains.** The *top figure* shows the engineering time saved in model optimization through OPTIKIT vs internally estimated human hours. In the *bottom figure*, the optimal TPS (Transactions Per Second, i.e., Throughput) after the OPTIKIT cycle has terminated vs the baseline. Results shown on three models.

In this paper, we introduce OPTIKIT, an automated LLM optimization framework designed to address these chal-

[1]eBay, Foundation Models Team, Amsterdam, the Netherlands
[2]Democritus University of Thrace, Xanthi, Greece. Correspondence to: Nicholas Santavas <nsantavas@ebay.com>.

Proceedings of the 9th MLSys Conference (Industry Track), Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

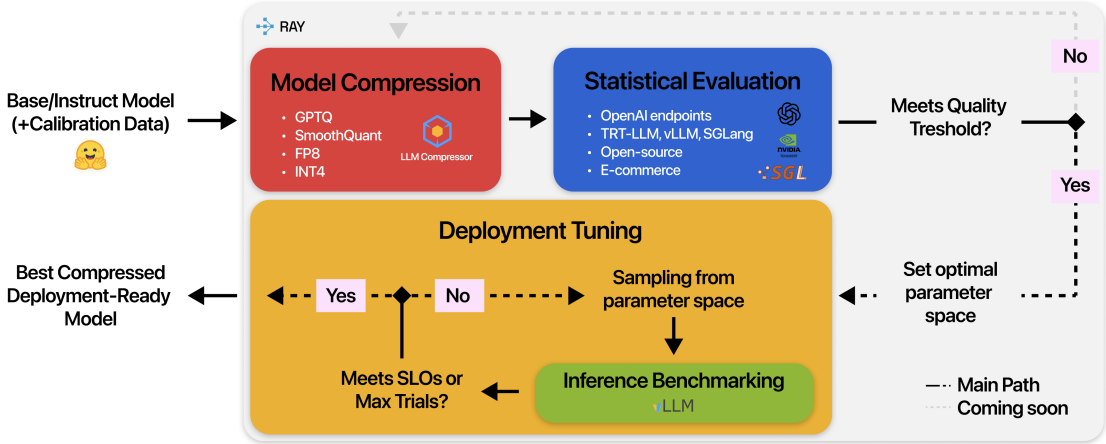


Figure 2: **OptiKIT full pipeline.** The figure shows the full OptiKIT flow. We begin by fetching any base/instruct model along with calibration data if needed, and apply model compression through the user-selected technique. We then proceed to perform a statistical evaluation of the optimized model to ensure the validity of our compression strategy. If the performance is up to standards, we determine the set of parameter space for deployment tuning. Subsequently, we sample from this space and perform Inference Benchmarking to determine the optimal subset of parameters for deployment. If the SLOs (Service Level Objectives) are not met, we iteratively repeat the above, sampling a new set of parameters. When a parameter configuration meets the SLOs, we return the model configuration along with its weights, ready for deployment. The small logos represent part of the back-ends supported.

lenges. Developed and deployed at eBay, OptiKIT embodies three core principles: automation, standardization, and deep enterprise integration. It provides a comprehensive, end-to-end solution that automates the optimization lifecycle, from model analysis and resource allocation to performance benchmarking and deployment. By standardizing this process, OptiKIT democratizes access to advanced optimization techniques, enabling any engineering team to achieve expert-level performance without requiring specialized knowledge.

Initial results at eBay demonstrate that OptiKIT can achieve significant throughput gains and latency reductions, enabling the deployment of more powerful models within existing resource envelopes (Figure 1). This suggests that a systematic, automated approach to LLM optimization is a technically feasible and critical component for enabling scalable, cost-effective AI in the enterprise (Zhen et al., 2025).

Our contributions can be summarized as follows:

- We present **OptiKIT**, a fully *automated end-to-end*, distributed, resource-aware LLM optimization pipeline with modular orchestration, dynamic GPU allocation, completely integrated into enterprise infrastructure.
- We **operationalize** this architecture through tailored algorithmic decision-making governed by strict Service Level Objectives (SLOs). Key mechanisms include: a backend-agnostic compression engine paired with a statistical evaluation mechanism acting as a quality-feasibility gate; regression-based stability detection for rigorous throughput certification under explicit serving profiles; and a Bayesian runtime tuner that performs sample-efficient search over costly trials, backed by complete artifact archival for auditability and replay.
- We conduct an **extensive empirical study** across *large-scale production workloads and model families*. OptiKIT achieves throughput gains of up to $2.8\times$ with robust, reproducible optimization across heterogeneous infrastructure.

2 The LLM Optimization Challenge at Scale

The deployment of Large Language Models in production environments presents a unique set of challenges that differ substantially from academic research settings (Chavan et al., 2024). Enterprise deployments must contend with hard constraints on GPU availability, heterogeneous hardware infrastructure, and the need for consistent performance across diverse workloads. At eBay, these challenges manifest in several key areas. First, the finite nature of GPU resources creates a zero-sum constraint where every inefficiency in one application directly impacts the capacity available for others. Hence, resource utilization efficiency is paramount. Second, the diversity of model architectures and use cases—ranging from 8B to over 70B parameters—demands flexible optimization approaches. Manual LLM optimization (Zhu et al., 2024) represents a significant organizational bottleneck. The process requires deep expertise in model compression,

Table 1: **OPTIKIT vs similar techniques.** We compare OPTIKIT with similar LLM optimization techniques.

Technique	Model Compression	Quality Gate	Inference Benchmarking	Deployment Tuning	Data Sovereignty	Production Ready
Manual optimization (Zhu et al., 2024)	✓	×	×	×	✓	×
TensorRT-Sweep (NVIDIA, 2024)	×	×	✓	✓	×	✓
GuideLLM (Neural Magic, 2024)	×	×	✓	×	✓	×
High-Throughput LLM Inference (Xiong et al., 2025)	×	×	✓	✓	×	×
SCOOT (Cheng et al., 2025)	×	×	✓	✓	×	✓
OPTIKIT (ours)	✓	✓	✓	✓	✓	✓

hardware-specific optimizations, and inference runtime tuning (Zhou et al., 2024) —knowledge that is concentrated among a small number of specialists. This expertise gap creates several problems: optimization work becomes a dependency that slows feature development; inconsistent approaches lead to suboptimal resource utilization; and the manual nature of the process introduces variability in outcomes.

We separate engine-specific tooling from the end-to-end deployment workflow. Engine-specific tools operate inside a particular serving stack (profiling, benchmarking, and parameter sweeps), while the workflow coordinates the steps that turn a base model into a deployment-ready, SLO-validated artifact (compression, quality checks, benchmarking, tuning, and archival). Table 1 compares native workflow capabilities; a ✓ indicates the tool provides that function itself rather than being wrapped by external orchestration (×).

Existing tools for LLM optimization, while powerful, often fall short in enterprise settings; see Table 1. A key distinction is *engine-layer* tooling (profilers, kernel/engine autotuners, and parameter sweeps) versus *workflow-layer* orchestration. Engine-layer tools (e.g., TensorRT-Sweep (NVIDIA, 2024)) can efficiently explore inference configurations, but typically do not provide end-to-end quality gating, SLO certification under an explicit serving profile, or complete archival of artifacts/metrics for auditability. Similarly, benchmarking frameworks (e.g., GuideLLM (Neural Magic, 2024)) provide valuable steady-state measurements, but they are not a substitute for a production workflow that (i) filters candidate models via statistical quality checks, (ii) searches for the maximum sustainable rate that satisfies deployment SLOs, and (iii) packages the resulting model, configuration, and evidence for rollout. OPTIKIT is designed to orchestrate this workflow; in this paper we use our own benchmarking stack (vLLM-based) and do not integrate external engine-layer tools.

These challenges underscore the need for a unified, production-grade optimization framework— a role OPTIKIT aims to fulfill.

3 SYSTEM DESIGN AND ARCHITECTURE

3.1 Design Philosophy

Motivation OPTIKIT follows three principles addressing the core challenges of large-scale LLM optimization.

Automation: End-to-end workflows for compression, calibration, and tuning are automated through declarative task definitions, ensuring reproducibility and consistency.

Resource Awareness: Heterogeneous resources are orchestrated according to each stage's compute and data characteristics, maximizing utilization and minimizing overhead.

Interoperability: Standardized interfaces connect to existing registries, data sources, and experiment tracking systems, enabling seamless integration into enterprise infrastructure.

Abstractions OPTIKIT represents each optimization operation as a single *process* (Figure 2), consisting of ordered stages that collectively form a streamlined flow:

1. **Fetch:** Retrieve the target model and, if provided, calibration data from remote storage to local workspace.
2. **Model Compression:** Parallelize quantization trials to capture variance in calibration data sampling.
3. **Statistical Evaluation:** Evaluate quantized models to measure accuracy and potential quality degradation.
4. **Inference Benchmarking:** Measure serving performance of the evaluated models under controlled load.
5. **Deployment Tuning:** Optimize runtime parameters such as parallelism, batch size, and context window.
6. **Upload:** Store the optimized model, associated metrics, and metadata to centralized tracking repositories.

Flows are defined declaratively, with OPTIKIT automatically mapping stages to actor pools and resource allocations at runtime to ensure reproducible, auditable execution.

Design Rationale Our architecture explicitly trades algorithmic novelty for system reliability, failure isolation, and automation. Rather than inventing new quantization mathematics, OPTIKIT relies on standard search heuristics (e.g., TPE, exponential bracketing) and off-the-shelf post-training quantization. This approach allows us to solve the primary enterprise bottleneck: operationalizing the workflow. For example, as shown in our abstractions, we intentionally

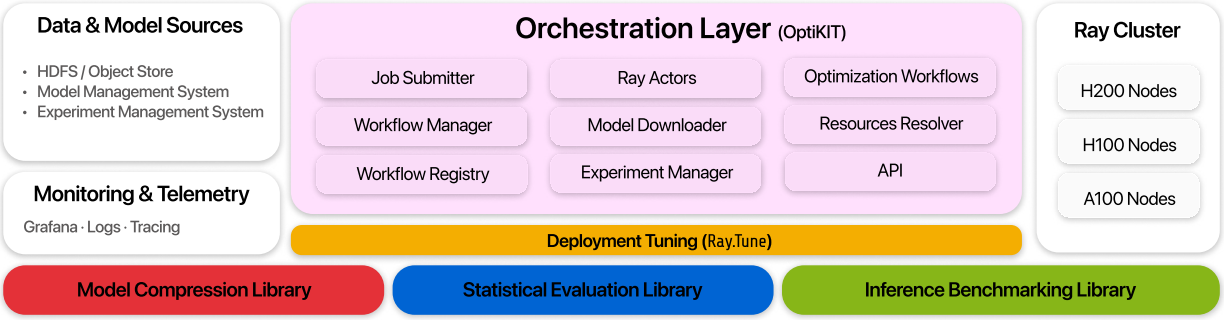


Figure 3: **OPTIKIT system architecture.** The orchestration layer manages submission, resource allocation, and experiment tracking via Ray Actors, integrating with external data/model sources and heterogeneous GPU clusters. Supporting libraries provide extensible optimization capabilities; monitoring ensures observability.

separate statistical evaluation from benchmarking. This is necessary because quality checks require variable-length I/O for correctness, whereas SLO certification requires tightly controlled, fixed traffic profiles.

3.2 Architecture Components

OPTIKIT is structured as a distributed Python SDK built on Ray (Moritz et al., 2018), organized into three fundamental architectural layers that provide clear separation of concerns and enable flexible, scalable optimization workflows.

Actor-Based Execution Layer The foundation layer consists of specialized Ray actors that handle specific optimization tasks. Each actor type encapsulates the logic for a particular operation and manages its own computational resources. Actors can be dynamically created with appropriate GPU/CPU allocations, scaled horizontally across the cluster, and terminated to free resources. This design enables fault isolation where individual actor failures don't compromise entire jobs, and efficient resource utilization through fine-grained allocation per computational profile.

Flow Composition Layer Flows implement the `BaseFlow` contract and compose low-level actors into executable pipelines. A flow is responsible for: instantiating and sizing actor pools for each stage, mapping declarative resource hints to concrete GPU/CPU allocations, queuing trial work and load-balancing it across available actors, and coordinating deterministic teardown to reclaim resources between stages.

We intentionally keep actor pools stage-isolated and ephemeral: compression, statistical evaluation, and benchmarking have different compute/memory footprints and failure modes, so per-stage pools allow tailored sizing and stronger fault isolation. Deterministic teardown also resets GPU state between stages, improving run-to-run consistency; the trade-off is barrier-induced idle time under stragglers, discussed in Section 6. Failure handling is explicit: transient actor errors trigger bounded retries, while persistent trial failures are recorded and excluded from further stages.

Each flow is bound to a versioned Docker image that encapsulates its runtime environment and dependencies, ensuring reproducibility and isolation across releases. Flows are registered via a Flow Registry, enabling teams to add new workflows without touching core runtime code. Each flow emits an archive of trial metadata, metrics, and artifacts to support reproducibility and post-hoc analysis.

Submission Engine Layer The submission engine manages job validation, packaging, and distributed execution. It converts high-level job specifications into executable configurations, validates them against `Pydantic` schemas, and bundles required artifacts—including the full OPTIKIT runtime—into a self-contained package for deployment. The engine coordinates authentication, resource allocation, and container orchestration through existing enterprise schedulers, providing a uniform interface for both local and remote execution.

4 CORE SUBSYSTEMS

OPTIKIT integrates several specialized subsystems that provide distinct optimization capabilities while maintaining seamless interoperability through standardized interfaces.

4.1 Optimizer: Universal Compression Framework

The *Optimizer* subsystem (Figures 2, 3) serves as OPTIKIT's universal engine for model compression (Zhu et al., 2024; Wang et al., 2024a; Zhou et al., 2024), providing a consistent interface for applying diverse optimization techniques across heterogeneous inference backends. Its backend-extensible design abstracts away engine-specific APIs behind a narrow contract, enabling portable compression workflows when adapters are provided for a given backend.

Backend-Extensible Architecture The *Optimizer* defines a standardized *Optimization-Backend* interface that decouples OPTIKIT flows from any specific compression library or inference engine. The contract exposes a small set of primitives (e.g., model load/save, strategy configuration, optimization execution), enabling alternative implementations to be swapped without changing workflow logic. In this paper, we instantiate the interface with our current production backend (integrating vLLM-compatible compression routines (Kwon et al., 2023; AI & vLLM Project, 2024)); supporting additional backends (e.g., TensorRT-LLM (NVIDIA Corporation, 2023)) requires implementing the same contract. Our evaluation uses vLLM for serving and benchmarking; other serving backends are not evaluated here.

Recipe-Based Configuration System The *Optimizer* introduces a *recipe-based configuration* paradigm that transforms model compression from ad-hoc tuning into a structured, declarative workflow. A *recipe* encodes a complete compression strategy—quantization scheme, calibration requirements, and layer-selection policy—into a reusable specification that captures domain heuristics such as layer exclusions and dataset size. Current recipes include:

- `int_w8a8` and `int_w4a16` — integer quantization recipes based on GPTQ (Frantar et al., 2023) and SmoothQuant (Xiao et al., 2023), representing robust post-training quantization and activation balancing.
- `fp8_dynamic` — a mixed-precision recipe derived from RTN (Micikevicius et al., 2022), suitable for layers sensitive to integer quantization.

This abstraction standardizes compression workflows across models and tasks while remaining easily extensible. New recipes can be registered to integrate emerging quantization methods or custom heuristics without modifying core optimization logic.

Calibration Data Sampling To support data-aware quantization (Xiao et al., 2023; Frantar et al., 2023), the *Optimizer* includes a modular sampling pipeline for calibration dataset preparation. As established in prior work (Williams & Aletras, 2023; Zhang et al., 2025), selecting the correct calibration data affects the performance of the quantized model. Accordingly, the developed module supports multiple data calibration pipelines. These range from uniform random sampling to more advanced strategies such as length-weighted and token-statistics–stratified sampling. The goal is to account for variations in data distribution, dataset composition, and token-level characteristics. The module also provides hooks for easy extension with new strategies. Its flexible design further enables future adaptive calibration, where sampling dynamically adjusts to quantization performance.

Automated Optimization Workflow The *Optimizer* automates the full compression lifecycle, including model loading, calibration preprocessing, quantization, and model serialization. Calibration sample counts are derived directly from recipe specifications, and the pipeline orchestrates execution end-to-end, enabling fully automated, reproducible compression across backends.

4.2 StatEval: LLM Statistical Evaluation library

The *StatEval* package (Figures 2, 3) is a core component of the optimization framework, handling statistical evaluation of model performance across multiple inference backends.

Design and Integration *StatEval* is built with a modular architecture that cleanly separates model handling from evaluation logic. The package is designed for seamless integration with internal eBay infrastructure while remaining easily adaptable to open-source and external environments. It supports two primary runtime interfaces: local vLLM for offline evaluation and OpenAI-style HTTP endpoints for online inference. Engines that expose OpenAI-compatible APIs (e.g., vLLM or SGLang (Zheng et al., 2024)) can be integrated with minimal changes; other engines require adapters. In this paper, we use vLLM for offline evaluation.

Within the workflow, *StatEval* acts as a quality-feasibility gate before performance search. We keep this stage separate from benchmarking because quality runs prioritize task-faithful prompts and scoring pipelines, whereas serving certification requires controlled arrival processes; conflating them is methodologically nonsensical, since the measured performance would primarily reflect benchmark-harness characteristics rather than deployment traffic and SLO behavior (see Section 4.3).

Supported Benchmarks *StatEval* is an internal package tailored for e-commerce–specific model evaluation. The package includes open-source benchmarks to enable standardized and comparable assessment: *GSM8K* (Cobbe et al., 2021), *IFEval* (Zhou et al., 2023), *Do-Not-Answer* (Wang et al., 2024b). The selected benchmarks cover three core LLM capabilities—reasoning, instruction-following, and safety—essential for both general use-cases and e-commerce applications. Additionally, we develop in-house benchmarks that are a core component of the system. They act as critical proxies of e-commerce production metrics for rapid experimentation cycles. These benchmarks are proprietary and excluded from this study.

4.3 Benchmarker: Performance Testing Tool

The *Benchmarker* library (Figures 2, 3) quantifies the performance capabilities of an optimized model while ensuring compliance with predefined SLOs such as end-to-end la-

tency and time per output token. Its purpose is to determine the maximum sustainable request rate that maintains target SLOs under a fixed serving profile Π. Because Π encodes request mix and input/output token-length characteristics, certified throughput is workload- and SLO-dependent rather than model-intrinsic.

The *Benchmarker* executes controlled load experiments on the optimized model within the Ray cluster, monitoring fine-grained telemetry via integrated tracing and metrics pipelines. These metrics serve as a critical interface between optimization workflows and deployment configurations, i.e., guaranteeing the required performance envelope for production rollout.

We structure SLO-gated benchmarking as a low-cost closed-loop infeasibility gate followed by open-loop certification: the first rejects configurations that fail even without queueing pressure, while the second measures sustainable throughput as batching and queueing emerge under Π. Moreover, by standardizing the SLO-driven benchmarking process, we ensure comparability across experiments, hardware classes, and compression strategies. The full procedure is summarized in Algorithm 1, which outlines the iterative search and decision logic governing the sweep. Below, we dive into core algorithmic components.

Steady-State Regression Assessing whether the system has reached steady state involves modeling the relationship between *request arrivals* and *completions* as a linear process. Compared to using a single, statically configured warmup time, this regression-based criterion provides a simple robustness check: it can detect when a trial is still "settling" (e.g., delayed initialization, lazy CUDA graph capture, or transient stalls) even if the run has already passed a fixed warmup period. In our implementation, we still apply a short fixed warmup to amortize one-time initialization overheads, and then use the regression signal during the measurement window as the accept/reject gate for stability. Specifically, we fit a regression of the form

$$r_i = \alpha + \beta\, c_i + \varepsilon_i, \tag{1}$$

where α is a fixed overhead, r_i denotes the arrival timestamp and c_i the completion timestamp of request i (both normalized relative to the start time), and ε_i captures residual noise. The estimated slope β acts as a compact indicator of system equilibrium:

- $\beta \approx 1$: **Steady-state** — completions keep pace with arrivals, and the queue length remains stable.
- $\beta > 1$: **Overloaded** — arrivals faster than completions, leading to backlog accumulation and latency inflation.

A trial is considered stable if $|\beta - 1| \leq \tau_\beta$, where τ_β is a small tolerance (typically 0.02–0.05 depending on noise and sampling granularity). In addition to the slope, the regression intercept and correlation coefficient are logged as part of the diagnostic record, providing visibility into drift patterns and fit quality. Figure 4 illustrates a typical diagnostic plot. A fitted slope near 1 indicates arrivals and completions are matched. Deviations from 1 reveal rate imbalance and queue growth, enabling the Benchmarker to find the highest sustainable load before instability.

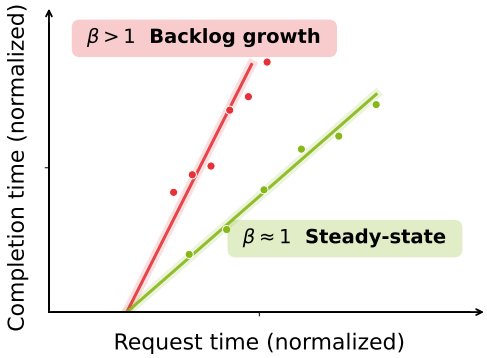


Figure 4: **Regression diagnostics for a Benchmarker trial.** A fitted slope $\beta \approx 1$ (*green*) indicates steady-state operation queuing while $\beta > 1$ (*red*) denotes an overload regime.

Beyond a short fixed warmup, we apply a regression-based stability check during the measurement window to handle variable initialization (e.g., lazy CUDA graph capture, dynamic batching warmup). This approach:

- **Detects stability violations:** By monitoring $\beta \approx 1$ (Eq. 1), we reject runs corrupted by insufficient warmup or runtime stalls.
- **Ensures data quality:** A tight tolerance ($\tau_\beta = 0.02$) prioritizes clean measurements. Our 900s trials ($\sim$thousands of requests) ensure statistical fit quality.

Exponential Search *Benchmarker* explores the feasible operating region through an adaptive sweep over candidate request rates. Each trial is executed at a fixed request rate and evaluated against both the SLO criteria and the steady-state condition described above.

Rates that satisfy all constraints are marked as *passing*, while those that violate any constraint are marked as *failing*. After each evaluation, the next test rate is selected using a *bounded search heuristic* that incrementally narrows the feasible region. We use exponential bracketing to quickly localize the feasibility boundary, then midpoint refinement to approximate the maximum SLO-feasible rate with logarithmic trial growth; this is more sample-efficient than dense fixed-grid sweeps when each trial is expensive.

If this baseline already violates the specified SLOs, the configuration is deemed infeasible and the sweep termi-

nates early. The final output comprises the highest passing rate—the system's sustainable throughput under SLO compliance—together with a complete archive of all tested rates, stability diagnostics, and latency statistics. This archive supports reproducibility, post-hoc analysis, and cross-hardware comparability across optimization trials.

We employ a two-phase approach to efficiently certify SLO compliance:

- **Phase 1 (Closed-loop infeasibility gate):** A low-cost check that rejects configurations failing the latency SLO even under zero queueing pressure. This eliminates obviously infeasible configurations without expensive rate sweeps.
- **Phase 2 (Open-loop certification):** An asynchronous constant-rate trial that measures sustainable throughput under a fixed serving profile Π. By fixing the arrival process, we inherently capture the emergent batching, queue dynamics, and stability as request rate r increases—without analytically "correcting" for batching effects.

This structure bounds search cost via early rejection while ensuring that certified configurations can sustain their claimed throughput under realistic queueing and batching behavior.

Algorithm 1 uses an adaptive search to efficiently pinpoint the system's saturation boundary. It implements exponential doubling ($2r$) of the request rate upon success to rapidly explore capacity limits, and dynamically switches to midpoint halving (bisection) upon failure to refine the exact threshold. This achieves logarithmic complexity, avoiding the prohibitive compute costs of exhaustive grid search while preventing premature convergence.

Traffic Pattern. The open-loop trials use a Poisson arrival process with mean rate r requests/second. Each request draws input/output lengths from the workload-specific distribution defined in Π (see Section 5 for per-workload profiles). This asynchronous pattern allows queue buildup and realistic batching to emerge naturally as r approaches the saturation point.

4.4 Tuner: Automated Hyperparameter Optimization

The tuning stage (Figures 2, 3) optimizes the runtime configuration of the quantized model to maximize inference throughput while maintaining compliance with SLOs. Rather than modifying model weights, it searches over inference engine runtime parameters that control parallelism, batching, and context allocation. Quality constraints are enforced by the preceding StatEval gate; incorporating accuracy explicitly into the tuning loop is straightforward by

Algorithm 1 *Benchmarker* Sweep (closed-loop infeasibility gate + open-loop certification under fixed profile Π)

Require: Serving profile $\Pi = \langle$input/output length distributions, request mix, think time, $\ldots\rangle$
(Optional) SLOs $\mathcal{S} = \{\}$, Error margins $\mathcal{E} = \{\}$
(Defaults) initial rate r_0, budget N, threshold $\mathcal{T}$

```
best ← none
if SLOs provided then
    Run a synchronous closed-loop trial under Π
    if ∃s ∈ S ∼ E violated then
        return ⟨status : INFEASIBLE, rate : 0.0⟩
    else
        LB ← E[latency^-1] (lower bound rate)
    end if
end if
r ← r_0
while not converged and no. trials ≤ N do
    Run an asynchronous open-loop trial at r under Π
    if ∀s ∈ S ∼ E passed and queuing steady-state then
        best ← r
        r ← 2r (exponential doubling)
    else
        r ← (LB+r)/2 (midpoint halving)
    end if
    if |best − r| ≤ T then
        set converged
    end if
end while
return ⟨status : FEASIBLE, rate : best⟩
```

adding quality metrics to the reported objective and treating them as additional constraints (or Pareto dimensions) alongside latency. Similarly, decision variables such as speculative decoding or MoE expert-parallel configuration can be surfaced as additional tuning dimensions whenever the serving backend exposes them; in this paper we focus on the core vLLM runtime knobs available in our deployment setting. The `TunerActor` integrates with the `BenchmarkerActor` subsystem to evaluate each candidate configuration under realistic serving workloads. Every Ray Tune trial executes a complete benchmark evaluation, measuring throughput, latency, and SLO compliance.

Optimization Objective The optimization objective combines these metrics into a single scalar fitness function, defined as:

$$\text{fitness}(c) = \frac{\text{throughput}(c)}{\text{tensor_parallel_size}(c)} + \lambda \cdot \text{slo_penalty}(c) \quad (2)$$

where c denotes a candidate configuration. Throughput is normalized per GPU to ensure fair comparison across different parallelization strategies, and λ applies a large negative

Algorithm 2 Quantization with Tuning Flow (trial-parallel, actor-pool based)

Require: Model $\mathcal{M}$, dataset $\mathcal{D}$ (optional), number of trials N_{trials}, resource budget R
Fetch $\mathcal{M}$ and $\mathcal{D}$ from remote storage; store locally
Sample N_{trials} distinct calibration subsets $\{\mathcal{C}_i\}_{i=1}^{N_{\text{trials}}}$
results $\leftarrow \varnothing$
create quantization actor pool sized to R
for all $i \in \{1, \ldots, N_{\text{trials}}\}$ **in parallel do**
  Apply quantization recipe to $\mathcal{M}$ with calibration $\mathcal{C}_i$; produce compressed model q_i
  Attach metadata (seed, recipe, path) and append $\langle \mathcal{C}_i, q_i \rangle$ to **results**
end for
destroy quantization actor pool {free GPUs and reset distributed state}
create evaluation actor pool sized to R
for all each compressed model q in **results in parallel do**
  Run statistical evaluation on q; attach quality metrics to its record
end for
destroy evaluation actor pool
Let $\mathcal{S}$ be successful candidates
if $\mathcal{S} = \varnothing$ **then**
  return failure status and archive
end if
Select representative quantized model $q^* \in \mathcal{S}$
create benchmarking actor pool sized to R
Benchmark q^* (and optionally full-precision baseline) to collect runtime + stability metrics
destroy benchmarking actor pool
Create single CPU tuning orchestrator
Build tuning search space (tensor parallel sizes, max_num_seqs, max_num_batched_tokens, ...)
for all configuration c proposed by tuner (Ray Tune / Optuna) **in parallel or sequential as resources permit do**
  Instantiate benchmark job for (q^*, c)
  Measure metrics (throughput, normalized request rate, pass_slo, etc.)
  Report metrics back to tuner
end for
Destroy tuning orchestrator
Persist: quantized model q^*, best tuning configuration c^*, metrics, and trial archive to EMS / model registry
return $\{q^*, c^*, \text{trial archive}\}$

penalty for SLO violations (typically $\lambda = -1000$). This formulation guides the search toward configurations that sustain high per-GPU throughput while satisfying latency and stability requirements. Our formulation maximizes throughput subject to a strict latency SLO. This aligns with production goals: serving latency acts as a binary constraint, meaning that once the target is met, system efficiency is strictly governed by throughput maximization.

Tuning Orchestration The tuning process is orchestrated by a single `TunerActor`, which builds its parameter search space using the same input and output configuration applied during benchmarking of the quantized model. This ensures that the tuning trials explore serving parameters under identical workload conditions, preserving consistency in sequence lengths, token limits, and request patterns. Each tuning trial spawns a temporary `BenchmarkerActor`, which launches a vLLM server, generates synthetic request batches, and runs steady-state load tests to measure request rate and SLO pass ratio.

The search explores key inference engine parameters that influence runtime efficiency:

- **KV-cache / max-context provisioning:** We tune the `maximum context size` (i.e., the KV-cache provisioning target) computed from user-specified input and output length requirements as $(input_len + output_len) \times 1.15$. Over-provisioning reserves additional KV memory and can reduce the feasible concurrency ceiling, while under-provisioning risks truncation or OOMs.
- **Parallelism Strategies:** The search space for `tensor` and `data` parallelism explores configurations such as $\{1, 2, 4, 8\}$ bounded by cluster resource availability and user-defined limits.
- **Batch Processing:** Other parameters such as `maximum concurrency` and `maximum token batch size` are tuned within user-configurable ranges or system defaults.

Ray Tune employs the Optuna (Akiba et al., 2019) search algorithm, which implements Tree-structured Parzen Estimators (TPE) (Watanabe, 2023) for Bayesian optimization. This strategy models the objective landscape probabilistically and selects configurations that balance exploration of new regions with exploitation of known high-performing areas, improving sample efficiency compared to random or grid search. Each configuration is benchmarked using the same load generation and measurement logic as the performance stage, and metrics are reported back to Ray Tune. The best-performing configuration, its associated metrics, and the full archive of evaluated trials are stored with the quantization artifacts and uploaded at the final pipeline stage.

4.5 Final Algorithm

Algorithm 2 integrates the full OptiKIT *Quantization with Tuning Flow* (elaborating on Figure 2) into a unified, resource-aware pipeline. The flow executes as a sequence of distributed stages; computationally independent stages are

parallelized across actor pools sized to available GPU/CPU resources. Each actor processes one trial at a time, and results are synchronized before proceeding. Actor pools are explicitly destroyed between stages to free GPU memory and reset distributed state, ensuring deterministic resource reclamation and reproducible runs.

5 EXPERIMENTS

5.1 Experimental Setup

For our experiments, we used NVIDIA H100 GPUs for both quantization and inference tuning tasks. Each experiment was executed within the same environment to ensure consistency and comparability across models and configurations. Through this setup, we evaluated both the statistical performance recovery of quantized models and the inference performance gains achieved through runtime tuning. The end-to-end optimization runtime along with the GPU Hours for each evaluated model and stage are summarized in Figure 5 and in Paragraph 5.4.

5.2 Evaluated Models and Configurations

We tested three open-source LLMs representative of different operational scales and latency requirements: Qwen 2.5 7B Instruct (Qwen et al., 2025), Mistral Small 3 24B Instruct (Mistral AI Team, 2025), and Meta Llama 3.3 70B Instruct (Aaron Grattafiori, 2024).

For each model, we applied three quantization recipes available in OPTIKIT: Dynamic FP8, INT W8A8 (static-weight / dynamic-activation), and INT W4A16 (static-weight / high-precision activation). For the INT-based configurations, we used the default calibration dataset from (Magic, 2024), performing five independent trials with 256 random calibration samples for W8A8 and 512 for W4A16 per trial. The FP8 configuration required no calibration data and therefore exhibits no trial variance.

This setup enabled a direct comparison between quantization precision, calibration strategy, and runtime optimization under realistic production-style workloads.

5.3 Statistical Performance

We used OPTIKIT to measure the impact of different quantization recipes on each model's statistical performance. Table 3 reports the best-performing trial result, the corresponding recovery ratio relative to the full-precision baseline, mean, standard deviation (STD) and relative standard deviation (RSD) across five trials. For GSM8K, we report 8-shot exact match in the Chain-of-Thought setting; for Do-Not-Answer, the harmless responses proportion; and for IFEval, the mean of prompt- and instruction-level accuracy, following Meta's recipe (Aaron Grattafiori, 2024).

Table 2: **Example Inference Use-cases.** Representative eBay-derived inference use cases, with model scale, input/output token ratios, and corresponding latency SLOs.

Model	Input / Output (prefix)	Performance Objective
Llama 3 70B	5000 / 500	Throughput-oriented
Qwen 2.5 7B	1200 / 80	Latency p95 $\leq$ 500 ms
Mistral Small 3 24B	3000 / 200 (2000)	Latency p95 $\leq$ 1500 ms
Mistral Small 3 24B	1500 / 1500 (1000)	TTFT p50 $\leq$ 50 ms, TPOT p50 $\leq$ 10 ms

Across all evaluated tasks, both FP8 Dynamic and INT W8A8 quantization achieved near full-precision performance, with average recovery rates exceeding 99%. For Qwen 2.5 7B, performance degradation was minimal—typically below 0.5%, and in some cases, quantized models slightly surpassed full-precision baselines, indicating robustness to reduced precision. Similarly, Mistral Small 3 24B retained strong accuracy, with FP8 and INT8 models maintaining within 1% of the original results on average. However, INT8 showed higher variability across tasks (RSD up to 1.4%), reflecting task-dependent sensitivity. Mistral exhibited the greatest degradation on the IFEval task (93.5% recovery), showing reduced ability to follow multiple instructions simultaneously compared to the full-precision counterpart. For Llama 3.3 70B, quantization maintained near-identical performance to full precision on GSM8K and IFEval (usually with recovery greater than 100%), while Do-Not-Answer exhibited a modest reduction to around 95% recovery.

Overall, FP8 and INT8 quantization effectively preserved model performance with minimal loss, whereas INT4, while viable in some cases, exhibited inconsistent behavior and greater sensitivity to task characteristics.

Run-to-run Consistency and Determinism Production settings place particular emphasis on run-to-run consistency and deterministic behavior, since evaluation results often drive automated model selection decisions. However, achieving strict determinism in LLM inference is challenging due to non-determinism in model execution and CUDA kernels. Table 4 presents results from 100 runs, illustrating variability under default vLLM settings versus deterministic mode (Kwon et al., 2023). Although disabling multiprocessing yields nearly deterministic results, it prevents deloading VRAM in one Python interpreter session and is only applicable to offline inference, thus imposing practical limitations.

When full determinism cannot be achieved, it is crucial to assess whether observed differences are statistically significant—especially when evaluation results guide automatic model selection, where random variation may lead to incorrect conclusions. Addressing this requires controlled evaluation protocols and statistically grounded comparison methods to ensure robust, reliable assessment, which will be a focus of future development.

Table 3: **Statistical performance across models.** Performance recovery and trial variance across quantization recipes, models, and benchmarks.

	Full precision	FP8 Dynamic		INT W8A8				INT W4A16			
Task	**Result**	**Result**	**Recovery**	**Result**	**Recovery**	**Mean**	**STD (RSD %)**	**Result**	**Recovery**	**Mean**	**STD (RSD %)**
Qwen 2.5 7B Instruct											
GSM8K	0.826	0.818	99.031%	0.823	99.637%	0.821	0.003 (0.365%)	0.807	97.7%	0.811	0.005 (0.617%)
IFEval	0.773	0.758	98.06%	0.767	99.224%	0.764	0.003 (0.393%)	0.795	102.846%	0.761	0.02 (2.628%)
Do-Not-Answer	0.970	0.972	100.206%	0.973	100.309%	0.973	0.001 (0.103%)	0.967	99.691%	0.967	0.002 (0.207%)
Mistral Small 3 24B Instruct											
GSM8K	0.868	0.864	99.539%	0.879	101.267%	0.876	0.007 (0.799%)	0.873	100.576%	0.862	0.008 (0.928%)
IFEval	0.784	0.777	99.107%	0.733	93.495%	0.718	0.01 (1.393%)	0.780	99.49%	0.776	0.005 (0.644%)
Do-Not-Answer	0.945	0.946	100.106%	0.936	99.048%	0.941	0.004 (0.425%)	0.946	100.106%	0.952	0.004 (0.42%)
Llama 3.3 70B Instruct											
GSM8K	0.914	0.915	100.109%	0.909	100.219%	0.909	0.005 (0.55%)	0.907	99.234%	0.908	0.001 (0.11%)
IFEval	0.912	0.920	100.877%	0.912	101.206%	0.915	0.005 (0.546%)	0.915	100.329%	0.917	0.004 (0.436%)
Do-Not-Answer	0.995	0.949	95.377%	0.948	95.578%	0.949	0.003 (0.316%)	0.951	95.578%	0.946	0.003 (0.317%)

Table 4: **Qwen 2.5 7B Instruct *without* vs. *with* determinism.** We report results for Qwen 2.5 7B Instruct (100 runs per task) *without* vs. *with* vLLM deterministic setting. The deterministic configuration yields nearly identical results across trials.

Tasks	**Statistics**			
	Min	Max	Mean	STD (RSD %)
Non-deterministic setting				
GSM8K	0.818	0.829	0.824	0.002 (0.243%)
IFEval	0.754	0.781	0.767	0.005 (0.652%)
Deterministic setting				
GSM8K	0.822	0.822	0.822	0.000 (0.0%)
IFEval	0.771	0.775	0.773	0.001 (0.129%)

5.4 Inference Performance

We next examined the impact of quantization and runtime tuning on inference efficiency. Each workload configuration in Table 2 represents a characteristic operational regime—varying in input–output token ratios, SLOs, and model scale—to reflect eBay's production inference patterns.

For each workload, we performed a controlled benchmarking study to disentangle the contributions of quantization and runtime tuning. The baseline used the FP16 model with default vLLM parameters. The quantization-only setup applied model compression while keeping vLLM defaults, isolating quantization effects. The tuning-only setup optimized the FP16 runtime configuration using OptiKIT's deployment tuner. Finally, the end-to-end configuration combined both quantization and tuned vLLM parameters to assess their joint impact. Each tuning study employed TPE optimization over 30 trials, jointly searching `max_num_seqs`, `max_num_batched_tokens`, and `tensor_parallel_size`.

The values in Table 5 represent normalized per-GPU throughput for configurations that meet their respective SLOs. We treat SLO compliance as a hard deployment constraint and optimize within the feasible set, rather than trading off SLO violations for higher raw throughput. This normalization enables direct comparison across tensor-parallel regimes that may use different GPU counts and highlights the most cost-effective configuration—i.e., the setup yielding the highest SLO-compliant throughput per GPU. Notably, for Qwen, SLOs were not met without either quantization or tuning at TP=1, while for Mistral, SLOs were not satisfied under TP=1 or TP=2 in the absence of these optimizations.

OptiKIT Computational Cost Analysis In Figure 5 we report, side by side, the total wall-clock OptiKIT runtime (5a) and the total GPU-hours (5b). On the left, Figure 5a reports the total time it takes to run OptiKIT per stage, regardless of the number of GPUs utilized. On the right, Figure 5b instead breaks down GPU-hours, accounting for parallelization, the number of GPUs used per trial, and computational overheads caused by scheduling inefficiencies. Deployment tuning constitutes the primary overhead, accounting for an average of 70% of the total runtime. In contrast, statistical evaluation requires a negligible fraction of the overall time ($\leq 0.15\%$).

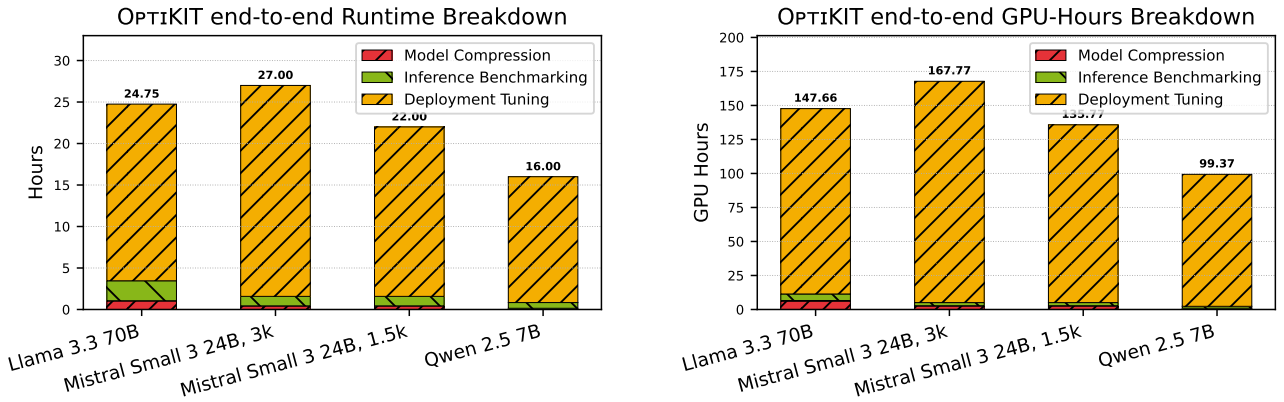
6 Discussion & Insights

Generalization of Quantization Quality Our results indicate that automated quantization with a generic calibration dataset (Magic, 2024) achieves stable and robust, production-ready quality without expert supervision (Table 3). This successful generalization, however, raises new questions about its boundaries. It is unclear if this robustness would hold after domain-specific fine-tuning (e.g., LoRA), or if domain-aligned calibration data would become necessary. Furthermore, the impact of short-context calibration on long-context task fidelity, which was not evaluated, remains

Table 5: **Normalized per-GPU throughput and improvement vs. FP16 baseline.** Normalized throughput per GPU (SLO-compliant); improvements as multiplicative factors vs. baseline. (*) marks SLOs not met without optimization.

Model	Baseline	Quantization only		Tuning only		Quantization + Tuning	
		Norm. TPS	Gain	Norm. TPS	Gain	Norm. TPS	Gain
Qwen 2.5 7B	3.52*	5.96	1.69×	6.79	1.93×	**7.50**	**2.13**×
Mistral 24B (Latency p95)	0.604*	1.732	2.87×	0.937	1.55×	**1.734**	**2.87**×
Mistral 24B (TTFT & TPOT p50)	0.562*	0.562	1×	0.750	1.33×	**0.875**	**1.55**×
Llama 3 70B	0.468*	**0.593**	**1.26**×	0.468	1×	0.585	1.25×

*SLOs not met without either quantization or tuning at the indicated tensor-parallel levels (TP=1, 2).



(a) **Wall-clock runtime breakdown.** Total optimization flow time per model and stage.

(b) **GPU-hours breakdown.** Total GPU-hours consumed across all stages. Considering parallelization, GPUs used per trial and scheduling issues.

Figure 5: **OptiKIT computational cost analysis.** (a) shows wall-clock runtime, while (b) presents total GPU-hours consumed. In both plots we report the breakdown per single Stage, Model Compression, Inference Benchmarking, and Deployment Tuning, with *StatEval* having negligible time ($\leq 0.15\%$) and therefore excluded. We report two Mistral Small 3 24B input/output scenarios as described in Table 2.

an open research question (Paglieri et al., 2024).

Tuning shines when SLOs are tight With strict SLOs (latency p95 and TTFT/TPOT), *tuning-only* improves FP16 by 1.33–1.55×. In multiple cases SLOs were unmet without optimization at lower TP, but became feasible after tuning and/or quantization (Tables 5, 7). When workloads operate near stability boundaries, the Benchmarker+Tuner (exponential search + TPE) finds SLO-compliant regions with higher sustainable rates, so relative tuning gains are largest in the most latency-critical production cases. In throughput-focused regimes, we observed diminishing returns, which warrant further investigation.

ROI of automation: Amortizing Siloed Efforts We quantify in Figure 1 the engineering cost of manual optimization, estimating it at 80–100 hours of specialized effort, compared to 15–25 hours for an automated OptiKIT run. For an industry setting, the primary contribution is not just the throughput gain but the drastic reduction in specialized, manual engineering cost. Manual optimization is a source of *hidden complexity* that creates knowledge silos, non-reproducible artifacts, and duplicated efforts across teams. By standardizing the process into a reproducible, end-to-end pipeline, OptiKIT democratizes performance tuning, enabling any application team to achieve expert-level optimization and directly addressing the organizational bottleneck of scarce ML systems expertise.

We also quantify computational efficiency: Figure 6 shows a mean GPU utilization of 88.0% during deployment tuning, with the 12% idle fraction stemming from (i) *within-wave straggler variance*, where some trials take longer to reach steady-state, leaving other GPUs idle at the barrier, and (ii) *last-wave under-utilization* when remaining trials are not divisible by available GPUs.

Risk Mitigation & Failure Modes OptiKIT addresses failure modes at multiple levels. First, the *StatEval* quality-gate (Figure 2) terminates pipelines whose quantized models

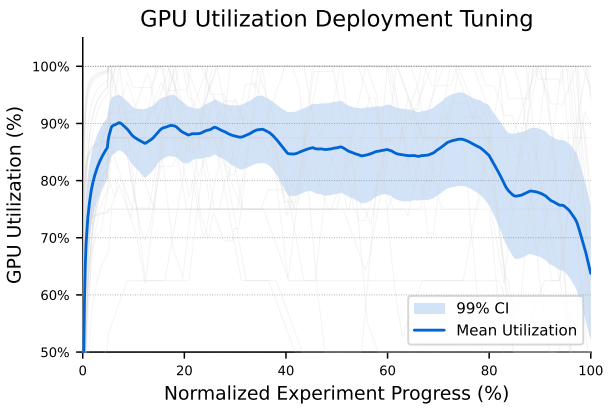


Figure 6: **GPU Utilization during Deployment Tuning.** Mean utilization across concurrent trials (88.0%); shaded region shows 99% CI. The drop near end highlights "last-wave" inefficiency from synchronization barriers.

fail accuracy checks before any expensive benchmarking or tuning begins; future versions will support automatic re-compression with varied calibration data or precision settings. Second, cost blowups under tight SLOs are controlled via (a) early infeasibility detection that quickly prunes unviable configurations, (b) capped exponential-bracketing and bisection within a fixed trial budget to efficiently locate the feasible performance boundary, and (c) an SLO-compliant objective normalized per-GPU to maximize throughput while strictly satisfying latency constraints. Third, tuning instability is handled on two fronts: the Benchmarker monitors steady-state regression signals (Eq. 1), immediately classifying configurations with growing backlogs as SLO-failing, while strict actor-pool isolation between stages prevents operational conflicts and guarantees a clean GPU state for each trial.

Since the pipeline archives all trial metrics, extracting a multi-objective Pareto frontier over cost, latency, and quality is a straightforward post-processing step. Teams can analyze trade-offs to select alternative configurations if business requirements change, avoiding the need to re-run expensive optimization sweeps.

Scope Limitations Our current evaluation covers dense decoder-only models with standard decoding and tensor parallelism up to 8 GPUs. We do not yet evaluate MoE-specific decision variables, speculative decoding parameters, or stateful multi-turn session dynamics (e.g., persistent KV-cache reuse and cross-turn dependencies).

7 Conclusion & Future Work

7.1 Conclusion

We present OptiKIT, an end-to-end, production-grade framework for automated LLM optimization with distributed and dynamic resource management. Unlike existing toolchains that address isolated aspects of the process, OptiKIT automates every stage—from model fetching and compression to statistical evaluation, inference benchmarking, and deployment tuning—enabling enterprise teams to obtain a production-ready optimized model with minimal manual intervention. Empirical evaluations across diverse model families and real-life production configurations demonstrate more than $2\times$ throughput improvements per GPU while maintaining near full-precision accuracy across reasoning, instruction-following, and safety benchmarks. These results validate OptiKIT as an effective and reproducible solution for large-scale, production-grade LLM optimization.

7.2 Future Work

Key directions for future development include: *(i) Advanced compression:* Integrating pruning (LeCun et al., 1989; Han et al., 2015; Zhang et al., 2023; Sun et al., 2024), structured sparsity (Ma et al., 2023; Mishra et al., 2021), and sparse-quantized representations (Dettmers et al., 2023) alongside quantization, which will require co-optimization strategies beyond the current sequential pipeline. *(ii) Asynchronous scheduling:* Replacing the current inter-stage synchronization barriers with an async task scheduler to improve GPU utilization and eliminate last-wave idle time (Section 4.5). *(iii) MoE and speculative decoding:* Extending to Mixture-of-Experts (expert parallelism, EP/TP co-tuning) and speculative decoding (draft model selection, acceptance policy), both requiring dedicated quality-latency-cost studies. *(iv) Multi-turn workloads:* Adding trace-driven, stateful benchmarking with cache-aware objectives that account for KV-cache residency, eviction pressure, and session-level tail latency.

8 Disclaimer

This research was conducted at eBay Inc. All intellectual property arising from this work is the sole property of eBay Inc. The external collaborator's involvement was limited to academic discussion and manuscript preparation and does not confer any ownership or intellectual property rights. All software, data, and methodologies described in this work were created under eBay's direction and within its research environment.

REFERENCES

Aaron Grattafiori, e. a. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

AI, R. H. and vLLM Project. LLM Compressor, 8 2024. URL https://github.com/vllm-project/llm-compressor.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.

An Yang, e. a. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Chavan, A., Magazine, R., Kushwaha, S., Debbah, M., and Gupta, D. Faster and lighter llms: A survey on current challenges and way forward. *arXiv preprint arXiv:2402.01799*, 2024.

Cheng, K., Wang, Z., Hu, W., Yang, T., Li, J., and Zhang, S. Scoot: Slo-oriented performance tuning for llm inference engines, 2025. URL https://arxiv.org/abs/2408.04323.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training Verifiers to Solve Math Word Problems, 2021.

Dettmers, T., Svirschevski, R., Egiazarian, V., Kuznedelev, D., Frantar, E., Ashkboos, S., Borzunov, A., Hoefler, T., and Alistarh, D. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*, 2023.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

Ma, X., Fang, G., and Wang, X. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720, 2023.

Magic, N. Llm compression calibration, 2024. URL https://huggingface.co/datasets/neuralmagic/LLM-compression-calibration.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., García, J., Micikevicius, V., Mirza, M., Subramanian, S., and Zhu, H. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*, 2022.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating sparse deep neural networks, 2021. URL https://arxiv.org/abs/2104.08378.

Mistral AI Team. Mistral Small 3, 2025. URL https://mistral.ai/news/mistral-small-3.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. Ray: A distributed framework for emerging ai applications. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, pp. 561–577, 2018.

Neural Magic, I. Guidellm: Scalable inference and optimization for large language models. https://github.com/vllm-project/guidellm, 2024.

NVIDIA. Tensorrt engine sweeping guide. https://docs.nvidia.com/deeplearning/tensorrt-cloud/latest/sweeping-engines.html, 2024. Accessed: 2024-04-15.

NVIDIA Corporation. Tensorrt-llm: High-performance inference for large language models. https://developer.nvidia.com/tensorrt-llm, 2023.

Paglieri, D., Dash, S., Rocktäschel, T., and Parker-Holder, J. Outliers and calibration sets have diminishing effect on quantization of modern llms. *arXiv preprint arXiv:2405.20835*, 2024.

Park, S., Jeon, S., Lee, C., Jeon, S., Kim, B.-S., and Lee, J. A survey on inference engines for large language models: Perspectives on optimization and efficiency. *arXiv preprint arXiv:2505.01658*, 2025.

Qwen, :, Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Tang, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., and Qiu, Z. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=PxoFut3dWW.

Wang, W., Chen, W., Luo, Y., Long, Y., Lin, Z., Zhang, L., Lin, B., Cai, D., and He, X. Model compression and efficient inference for large language models: A survey. *arXiv preprint arXiv:2402.09748*, 2024a.

Wang, Y., Li, H., Han, X., Nakov, P., and Baldwin, T. "do-not-answer: Evaluating safeguards in LLMs", March 2024b.

Watanabe, S. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *arXiv preprint arXiv:2304.11127*, 2023.

Williams, M. and Aletras, N. On the impact of calibration data in post-training quantization and pruning. *arXiv preprint arXiv:2311.09755*, 2023.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pp. 38087–38099. PMLR, 2023.

Xiong, Y., Huang, J., Huang, W., Yu, X., Li, E., Ning, Z., Zhou, J., Zeng, L., and Chen, X. High-throughput llm inference on heterogeneous clusters, 2025. URL https://arxiv.org/abs/2504.15303.

Zhang, Y., Zhao, L., Lin, M., Sun, Y., Yao, Y., Han, X., Tanner, J., Liu, S., and Ji, R. Dynamic sparse no training: Training-free fine-tuning for sparse llms. *arXiv preprint arXiv:2310.08915*, 2023.

Zhang, Z., Gao, Y., Fan, J., Zhao, Z., Yang, Y., and Yan, S. Selectq: Calibration data selection for post-training quantization. *Machine Intelligence Research*, pp. 1–12, 2025.

Zhen, R., Li, J., Ji, Y., Yang, Z., Liu, T., Xia, Q., Duan, X., Wang, Z., Huai, B., and Zhang, M. Taming the titans: A survey of efficient llm inference serving. *arXiv preprint arXiv:2504.19720*, 2025.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Zhou, J., Lu, T., Mishra, S., Brahma, S., Basu, S., Luan, Y., Zhou, D., and Hou, L. Instruction-Following Evaluation for Large Language Models, 2023.

Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Yuan, Z., Li, X., et al. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*, 2024.

Zhu, X., Li, J., Liu, Y., Ma, C., and Wang, W. A survey on model compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1556–1577, 2024.

A System Design and Architecture

A.1 Architecture Components

This appendix provides an illustrative view of the core OptiKIT runtime components. Each snippet corresponds to a minimal, self-contained example that demonstrates how optimization flows are constructed, submitted, and executed within the distributed optimization framework.

A.2 Code walkthrough and explanations

Optimizer & Recipe definition. The first example shows the high-level optimizer interface. The user selects a backend implementation (here the vLLM compressor) and instantiates a quantization recipe. Recipes encapsulate all quantization hyperparameters and produce a concrete execution strategy through the create() call. The pipeline then runs end-to-end—from model retrieval to compression and artifact generation—under a unified interface once a backend adapter is implemented.

```
# Unified interface across implemented
    backends
optimizer =
    Optimizer(LLMCompressorBackend())
# vLLM backend (used in this paper)

# Recipe bundles all optimization
    specifications
recipe = get_recipe("int_w8a8")
# W8A8 quantization recipe
strategy = recipe.create()
# Complete configuration generated

# Automated optimization pipeline
optimizer.run_pipeline(
    model_path="llama-70b/v1.0",
    output_path="./optimized_model",
    strategy=strategy
)
```

Flow definition. The next listing defines a registered flow responsible for orchestrating quantization trials. A flow coordinates distributed actors in Ray, creates resource-scaled pools for quantization and evaluation, and manages trial queues as resources become available. Each flow explicitly declares its required parameters, enabling validation and reproducibility at submission time.

```
@FlowRegistry.register("quantization")
class QuantizationFlow(BaseFlow):

    def run(self, job: OptimizationJob)
    -> Dict[str, Any]:

    # Create ActorPools with dynamic
    resource allocation
        quant_pool =
    self._create_quantization_actors(
            context
        )
        eval_pool =
    self._create_evaluation_actors(
            context
        )

    # ActorPools queue trials and
    consume them as actors become
    available
        self._run_quantization_stage(
            trials, quant_pool
        )
        self._run_evaluation_stage(
            trials, eval_pool
        )
        results = self._build_results(
            trials
        )
        return results

    @property
    def required_params(self) ->
    List[str]:
        return [
            "quantization_recipe",
            "num_trials"
        ]
```

Quantization Actor Each quantization actor performs one independent compression trial. It loads the model, applies the specified quantization recipe, and emits the path of the resulting optimized model. Actors are GPU-bound and execute in isolation, ensuring deterministic per-trial behavior and clean teardown between experiments.

```
@ray.remote
class QuantizationActor(BaseActor):

    def run(self, trial_id: str, config:
    QuantizationConfig):

        result = self._compress(
            config.model_path,
            config.quantization_recipe
        )

        return {"quantized_model_path":
    result}
```

Submission example. This submission example shows how an optimization job is described and dispatched. A job specification includes model metadata (from MMS), calibration dataset location, flow parameters such as the quantization recipe and number of trials, and hardware requirements. The submitter component serializes the configuration and triggers execution on the Ray cluster, returning structured results with metrics and artifact locations.

```
job = OptimizationJob(
```

```
    name="llama_70b-compression-job",
    flow="quantization",
    model=MMSModelConfig(
        repo="models",
        name="llama-70b",
        version="v1.0"
    ),
    dataset=HadoopDatasetConfig(
         hdfs_path="/data/calibration"
    ),
    flow_params={
        "quantization_recipe": "int_W8A8
    (Dynamic)",
        "num_trials": 5
    },
    compute_config=[

    ComputeConfig(sku=ResourceSKU.H100_8)
    ]
)

submitter = Submitter()
result = submitter.submit(job)
```

B EXPERIMENTAL RESULTS

All throughput measurements were collected using a steady-state inference benchmark based on the vLLM serving stack. Each configuration was tested under fixed input/output sequence lengths and latency SLOs as shown in the table headers. Reported values correspond to the normalized per-GPU TPS achieved while meeting the latency target. Runs lasted 900 s of requests submission per sweep to ensure stable utilization. Configurations that did not satisfy latency SLOs are marked as **.

Table 6: **Normalized per-GPU throughput for FP16 tuning across tensor parallelism levels.** Values are normalized per-GPU RPS (SLO-compliant). Gains are shown vs. FP16 baseline. Missing baselines (**) indicate configurations not measured or not SLO-compliant.

TP	Baseline (FP16)	FP16 (Tuned)	Gain (Tuned / Baseline)
Qwen 2.5 7B (Input 1200, Output 80, Latency P95 500 ms)			
1	—	—	**
2	3.52	5.12	1.45×
4	4.68	6.79	1.45×
Mistral Small 3 24B (Input 3000, Output 200, Prefix 2000, Latency P95 1500 ms)			
1	—	—	**
2	—	—	**
4	0.604	0.937	1.55×
Mistral Small 3 24B (Input 1500, Output 1500, Prefix 1000, TTFT P50 50 ms; TPOT P50 10 ms)			
1	—	—	**
2	—	—	**
4	0.562	0.750	1.33×

**SLOs not met or FP16 baseline unavailable for the given TP.

Table 7: **Normalized per-GPU throughput and improvement vs. FP16 baseline across models, tensor parallelism, and bitwidths.** Values are normalized per-GPU RPS (SLO-compliant). Missing baselines (**) indicate configurations not measured or not SLO-compliant.

TP	Bitwidth	Baseline (FP16)	Quantization only		Quantization + Tuning	
			Norm. TPS	Gain	Norm. TPS	Gain
Qwen 2.5 7B (Input 1200, Output 80, Latency P95 500 ms)						
1	FP W8A8 (Dynamic)	—	3.70	**	3.67	**
1	INT W8A8 (Dynamic)	—	3.66	**	3.67	**
1	INT W4A16	—	2.01	**	3.20	**
2	FP W8A8 (Dynamic)	3.52	5.96	1.69×	7.49	2.13×
2	INT W8A8 (Dynamic)	3.52	5.95	1.69×	7.50	2.13×
2	INT W4A16	3.52	5.15	1.46×	5.14	1.46×
4	FP W8A8 (Dynamic)	4.68	5.62	1.20×	5.60	1.20×
4	INT W8A8 (Dynamic)	4.68	5.62	1.20×	5.62	1.20×
4	INT W4A16	4.68	4.68	1.00×	4.68	1.00×
Mistral Small 3 24B (Input 3000, Output 200, Prefix 2000, Latency P95 1500 ms)						
1	FP W8A8 (Dynamic)	—	—	—	—	—
1	INT W8A8 (Dynamic)	—	—	—	—	—
1	INT W4A16	—	—	—	—	—
2	FP W8A8 (Dynamic)	—	—	**	0.614	**
2	INT W8A8 (Dynamic)	—	—	**	—	**
2	INT W4A16	—	0.335	**	0.617	**
4	FP W8A8 (Dynamic)	0.604	1.732	2.87×	1.734	2.87×
4	INT W8A8 (Dynamic)	0.604	1.523	2.52×	1.523	2.52×
4	INT W4A16	0.604	1.125	1.86×	1.523	2.52×
Mistral Small 3 24B (Input 1500, Output 1500, Prefix 1000, TTFT P50 50 ms and TPOT P50 10 ms)						
1	FP W8A8 (Dynamic)	—	—	—	—	—
1	INT W8A8 (Dynamic)	—	—	—	—	—
1	INT W4A16	—	—	—	—	—
2	FP W8A8 (Dynamic)	—	—	—	0.506	**
2	INT W8A8 (Dynamic)	—	0.221	**	0.506	**
2	INT W4A16	—	0.148	**	0.492	**
4	FP W8A8 (Dynamic)	0.562	—	—	0.875	1.56×
4	INT W8A8 (Dynamic)	0.562	0.531	0.95×	0.875	1.56×
4	INT W4A16	0.562	0.562	1.00×	0.875	1.56×

**SLOs not met or FP16 baseline unavailable for given TP.