Understanding and Exploiting Workload Similarity for Effective Tuning of LLM Serving

Paper #1258

Abstract

LLM serving engines expose various configuration knobs whose optimal setting improves per-GPU performance by up to 1,827% over the engine's default setting. The optimum is workload-specific, so each cluster must be tuned individually, yet existing approaches treat the tuning as a black-box optimization problem, lacking a principled understanding of when a tuned configuration remains valid, how to evaluate candidates accurately, and how to search the space efficiently.

This paper presents a principled tuning approach built on one key insight: given a hardware platform and engine implementation, serving performance is determined by three complementary workload features—per-request computation, inter-request KV-cache hits, and arrival dynamics. We capture these features in a compact, 10-dimensional scale-stationary vector and define a *workload similarity metric*: workloads similar under this metric share optimal configurations, so the metric tells us when retuning is needed and which evaluations are trustworthy. Meanwhile, the knobs that matter for a good configuration correlate strongly with the workload features, capturing the domain knowledge engineers apply when tuning manually. We validate this approach with a systematic characterization using one-week production traces from COMPANYX—one of the world's largest LLM providers—and confirm the effectiveness. Building on these findings, we build AITUNER, an automated tuner that encodes these correlations as natural-language pruning rules for an LLM to compose a reduced search space. AITUNER improves per-GPU performance by 9.5% on average over production baselines has much shorter tuning time versus existing or simple LLM-based automated tuners.

1 Introduction

Setup and motivation. Large language model (LLM) providers serve a diverse mix of applications—chat services [23], coding agents [15], and general-purpose APIs [22]—across models of varying sizes and architectures [14, 24], on heterogeneous hardware. To manage this diversity, providers organize their infrastructure into *serving clusters*, each dedicated to one model serving a specific application (e.g., chat or coding), running on a homogeneous set of GPUs with a single serving engine (§2).

The configuration of the serving engine has a large impact on cluster efficiency [37, 12]: our analysis in §3 shows that the best configuration improves per-GPU performance by 1,827% over the engine's default setting, translating to 94.8 % fewer GPUs for deploying such a cluster. Given the high cost of GPU clusters, this is a significant saving for model providers.

Finding a near-optimal configuration for each cluster is challenging. Modern serving engines expose various knobs—chunked prefill [8], batch sizes, and various degrees of parallelism [18, 20]—resulting in a combinatorial search space of over hundreds or even thousands of configurations (Figure 1 (d)). The optimal configuration is highly workload-dependent: the best configuration for a chat cluster degrades performance by 43.9% when applied to a coding cluster (§3). Each cluster must therefore be tuned individually, making a fast, automated tuning pipeline essential.

State of the art. To cope with this complexity, both industry and academia have converged on a standard offline, evaluation-driven tuning pipeline (Figure 1 (a)): ① define the tuning target (which workload to tune for), ② evaluate a candidate configuration, and ③ search for the next candidate, iterating until a budget is exhausted. Despite this common structure, a principled understanding is missing: *how do workload characteristics determine the right engine configuration on a given hardware?* Without this understanding, every step relies on heuristics:

① *When does a tuned configuration remain valid?* A configuration is tuned on one workload observed in the past [12] or a synthetic one defined by the developer [5, 37], but the future workload that deploys the tuned configuration may differ. Without understanding how much drift invalidates a configuration, practitioners either retune on a fixed schedule—wasting GPU hours when the workload has not changed—or retune too late, serving with a stale configuration for days.

② *How to accurately evaluate a candidate configuration?* Configuration tuning relies on evaluation to select candidates. Existing works adopt synthetic workloads [37] or replayed past traces [12] to evaluate a configuration, but it is still unknown whether such evaluation is accurate: the distribution and duration of the evaluated workload inevitably differ from the target future workload. An inaccurate evaluation misleads the search, causing the tuner to discard superior configurations or converge to one that underperforms in production.

③ *How to search efficiently?* Existing automated strategies—from heuristic rules [5] and grid search [37] to AI-driven optimization [12]—treat the configuration space as an opaque, high-dimensional landscape, requiring many evaluations to converge or risking missing better configurations. Human-in-the-loop tuning, as currently practiced at COMPANYX, does

leverage domain knowledge to navigate the space effectively, but demands significant engineering effort per cluster and cannot scale to hundreds of clusters with continually shifting workloads.

Key insight and methodology: workload similarity-guided approach. The three open questions above all depend on how a workload interacts with the serving engine, yet existing work treats workloads as opaque distributions without decomposing them into the features that determine the optimal engine configuration. Our key observation is that given a hardware platform and engine implementation, the serving performance is determined by three complementary workload features—per-request computation (token *lengths*, L), inter-request KV$ hits (*cache*, C), and *arrival* dynamics (A) (§4)—which can be captured by a compact, 10-dimensional feature vector. This low dimensionality makes it practical to define a *workload similarity metric* to systematically answer all three questions: (1) when the metric shows that a cluster's workload has not drifted in these features, its tuned configuration remains valid; (2) when an evaluation workload is similar to the target in these features, its ranking of configurations is trustworthy; (3) the feature dimensions that differ most between workloads reveal which configuration knobs need re-adjustment, reducing the search space.

We validate this insight and our metric through a systematic study of LLM engine configuration tuning on production cluster traces from COMPANYX—one of the world's largest model-as-a-service providers. Our study confirms that similarity-guided tuning is effective with the following three takeaways:

(§5.1) *Intra-cluster workloads are similar across days; cross-application transfer fails.* Across the production clusters we studied, which span three distinct application types, a configuration tuned on one day's trace performs within 5.5% of optimal on subsequent days. Transfer across clusters serving different applications, however, fails: a chat cluster and a coding cluster have fundamentally different request distributions, so a configuration tuned for one degrades performance by 43.9% on the other. Our similarity metric confirms this pattern: intra-cluster similarity remains high across adjacent days ($\approx$ 88% overall LCA similarity), while cross-application similarity is consistently low. Moreover, within a cluster the L-C-A profile is scale-stationary at finer time granularity, only total request volume varies—a dimension absorbed by cluster autoscaling rather than per-engine tuning—while all configuration-relevant features remain nearly unchanged.

(§5.2) *Evaluation accuracy is governed by workload similarity, not wall-clock time.* A few minutes of real-trace replay suffice to reliably rank configurations, because the L-C-A similarity between the replayed segment and the full trace converges quickly. Among the three dimensions, cache dynamics (C) converges slowest and is the primary source of evaluation error: synthetic and semi-real evaluators that omit C lose up to 35% on cache-heavy workloads. Augmenting a semi-real evaluator with cache-aware sampling (Semi-real +C) closes this gap to within 2% of real-trace tuning, providing a practical path for tuning when production traces are unavailable.

(§5.3) *Workload-feature–to–knob correlations compress the effective search space.* Although the full configuration space is combinatorially large, we find that the L-C-A dimensions that distinguish workloads also predict which knobs need adjustment. For example, coder and chat differ most in the L and C dimensions, and these differences predict which knob families require re-tuning. These feature-to-knob correlations capture, in structured form, the domain knowledge that experienced engineers apply when tuning manually, and can serve as pruning rules to compress the search space.

(§6) An effective tuner: AITUNER. The feature-to-knob correlations above are imprecisely specified domain knowledge—natural for engineers, but ill-formed for a conventional optimizer. To harness them, we build AITUNER, an automated tuning system that uses an LLM to bridge this gap: AITUNER is directly incorporated into the shadow traffic pipeline to preferentially use real traces for tuning, with two similarity-guided improvements: (1) during search, it encodes the correlations as natural-language *harnesses*—rules that map the current bottleneck regime to the relevant knob family—and injects them into an LLM agent's prompt, so the agent focuses on productive knob adjustments from the first trial; (2) during evaluation, it applies *adaptive stopping* by monitoring L-C-A similarity convergence in real time, terminating each trial once ranking accuracy stabilizes rather than running for a fixed window.

We evaluate AITUNER on 5 production cluster workloads collected from COMPANYX. AITUNER discovers configurations that improve per-GPU performance by 9.5% on average over the manually tuned baselines currently in production, with three out of five clusters seeing improvements of 13% or more. Compared to SCOOT and a naive LLM-based tuning loop, AITUNER finds better configurations on all five clusters with a much shorter tuning time.

We will open-source AITUNER and samples of evaluated (tuned) traces.

2 Background: LLM Serving

LLM basics: prefill, decode and KV$. LLMs generate tokens in an auto-regressive manner [29], so serving proceeds in two phases. *Prefill* consumes the full input prompt in a single forward pass, computing key-value cache (KV$) entries for every input token and producing the first output token; its cost is proportional to input length. *Decode* then emits one token per step, reusing the cached KV$ rather than recomputing it, until an end-of-sequence (EOS) token is produced. Because decode reads KV$ from the GPU memory instead of recomputing it as prefill does, the two phases have

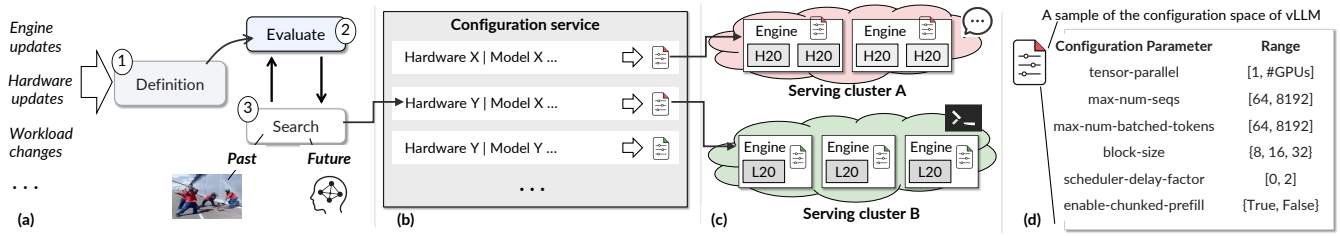

Figure 1: *(a) The configuration tuning pipeline. (b) The engine configuration service deployed in the wild. (c) An overview of serving LLMs handling multiple workloads. (d) Configuration entries to tune sampled from vLLM [6].*

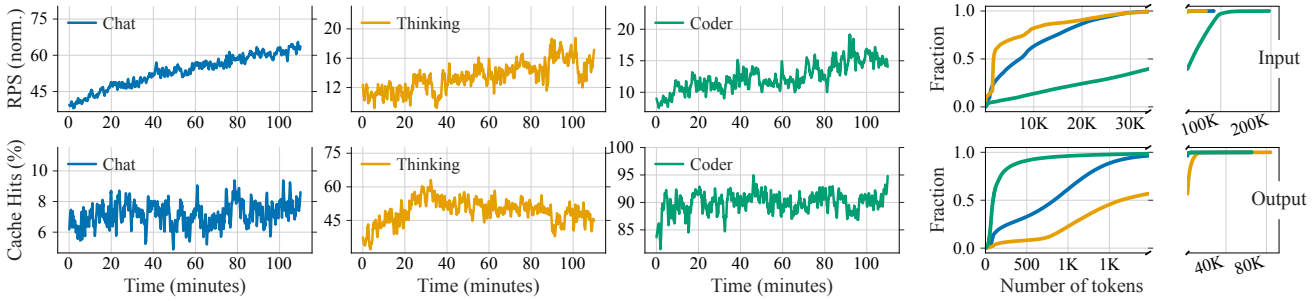

Figure 2: *A sample of our studied traces in* CompanyX.

distinct execution patterns. Because KV$ can be reused by requests with the same input, existing vendors cache them on the engines [30, 26], and a KV$ *hit* occurs when a prefix of the current request already has cached entries on the serving instance, allowing the engine to skip that portion of prefill.

Serving different LLM applications with different clusters. As noted in the introduction, each serving cluster is dedicated to one model, one hardware type, and one workload type (Figure 1 (c)). This is a coarse-grained partition: within a data center, there is typically one cluster for each (model, GPU type, workload type) combination. Within a cluster, the GPU fleet is partitioned into *instances*, each running a full model replica on a fixed set of GPUs using the same serving engine. A global router dispatches incoming requests across instances in a load-balanced and LLM-friendly way [40, 28]. The number of instances varies over time due to autoscaling to handle temporal workload variations[39, 35], though changes are infrequent: provisioning new GPUs takes time, and the provider minimizes GPU count per-cluster to reduce cost. Per-workload isolation avoids interference in batching and scheduling across workload types with different SLOs [42].

To simplify deployment, CompanyX maintains a *configuration service* (Figure 1 (b)) that maps each (model, hardware, SLO target, ...) tuple to a serving engine configuration. Today, these configurations are first hand-tuned by engineers and then pushed to the service. We seek to replace this manual process with an effective and efficient automatic tuning system.

3 Motivation: Optimizing Objective and the Configuration Matters

Performance objective. The deployment goal for a serving cluster is to minimize the number of GPUs needed to handle incoming requests while meeting service level objectives (SLOs). Throughout this paper, we use a length-aware TTFT target (stricter for shorter prompts and looser for longer prompts) plus a fixed TPOT target, following common practice in prior serving work [40, 33] and in CompanyX deployments. The TTFT target scales with input length because prefill time grows linearly with the number of input tokens, while the TPOT target remains constant because per-token decode latency remains stable under current optimizations [18, 38, 13].

Methodology of comparing two configurations. The most direct way to compare two configurations is to deploy each on a cluster matching the production scale, replay the full cluster workload, and compare the minimum number of GPUs at which each configuration meets the SLO for that traffic. However, this is impractical: even the smallest production cluster may far exceed the GPU resources available for the development testbed used for tuning and analysis.

Instead, we compare configurations using a metric we call *peak sustainable rate*: the maximum per-GPU request rate at which SLO attainment remains above 99%. Higher peak sustainable rate means fewer GPUs are needed: under the uniform dispatching of the global router (§2), each GPU in a cluster of N GPUs handles load W/N from total workload W, so the minimum GPU count is $N_{\min} = \lceil W \;/\; \text{peak sustainable rate} \rceil$—directly the cost-minimization objective above.

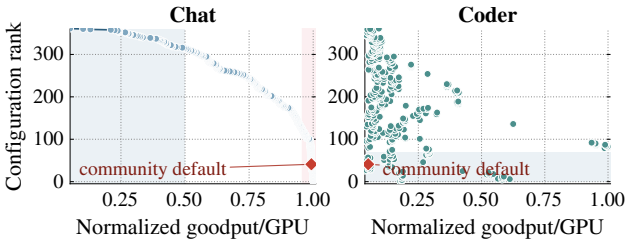

Figure 3: *Normalized goodput/GPU of Qwen-30B under 360 configurations for two workloads (left and right). Configurations are sorted by Chat goodput/GPU.*

We measure peak sustainable rate by sweeping the offered load—varying the sampling ratio of the production trace—and reporting the highest per-GPU request rate at which SLO attainment stays above the threshold. Each configuration is evaluated with a 5-minute trace replay; §5.2 provides evidence on why this duration is sufficient.

Evaluation setup: testbed, models evaluated and workloads. Unless otherwise stated, all experiments in this paper use a 32-GPU NVIDIA H20 testbed; in §7 we additionally include a cluster with 8 RTX 5090 to match the production setup (see §7).

To ensure our findings generalize across the diversity of production deployments, we use three representative workload traces collected from production clusters. All traces are anonymized and aggregated at the cluster level and contain no user-identifiable information. They cover three representative workloads: *Chat* (interactive conversation), *Thinking* (long-reasoning), and *Coder* (coding-agent). Figure 2 summarizes the workloads studied.

Because each production trace comes from a GPU fleet far larger than our testbed, we scale down the trace by randomly sampling requests at various testbed-to-production GPU ratios, following prior work [16, 25, 30].

Finally, unless otherwise noted, for each trace we run experiments on the same model and GPU hardware as the production cluster where the trace was collected, so that configurations operate under identical model–hardware constraints as the original deployment.

Configuration matters. Serving-engine configuration has a large impact on performance, and the right configuration differs across workloads. Figure 3 presents the normalized peak sustainable request rate of Qwen3-30B-A3B across 360 configurations for two production cluster workloads, Chat and Coder. We make the following observations: First, the performance differences across configurations are large: the best configuration outperforms the worst by 19 × on Chat and 26 × on Coder. Second, a configuration that is near-optimal for one workload can be far from optimal for another: vLLM configuration [6] (community default) achieves within 1% of the best goodput on Chat, yet delivers only 5% of the best on Coder.

More specifically, Figure 3 is drawn from a grid search of 360 configurations from vLLM [6]. Note that to make the search practical, we manually remove knobs that we are confident are unrelated to the performance. To visualize the different configuration behaviors across workloads, we explicitly use the same configuration rank across two experiments, i.e., configuration ranked i-th on Chat is also ranked i-th on Coder. The default configuration provided by the community [6] (a simple run under TP=1 without any specific flag) is also highlighted as a reference, which is near optimal for Chat but performs poorly on Coder.

These results motivate two requirements for our system. First, configuration tuning is essential: a poorly chosen configuration leaves more than an order of performance on the table. Second, there is no one-size-fits-all configuration—each workload demands its own tuning. §5.3 elaborates on why a configuration on Chat is insufficient on the Coder.

Table 1: *Mapping from workload dimensions to engine (vLLM) configuration knobs.*

Workload dimension	#Knobs	Knobs
(L) Input/output lengths	12	TP, EP, max-num-batched-tokens, cudagraph-capture-sizes, enable-chunked-prefill, ...
(C) KV$ hits	7	TP, gpu-memory-utilization, block-size, enable-prefix-caching, ...
(A) Arrival dynamics	7	DP, max-num-seqs, max-num-batched-tokens, ...

4 Workload Similarity: A Unifying Lens for Configuration Tuning

As we have mentioned in §1, existing tuning approaches treat workloads as opaque distributions, lacking a principled understanding of when a configuration transfers, when an evaluation is trustworthy, or which knobs matter. To close this gap, we introduce a *workload similarity metric*—the first, to our knowledge, to distill LLM serving workloads into the features that determine engine configuration. This metric guides our study of the effectiveness and efficiency of tuning in later sections.

4.1 What Determines Engine Configuration?

We systematically studied all performance-related configuration knobs exposed by vLLM 0.17.1 [6], which is also similar to the internal engines used at CompanyX. We find that each knob's optimal setting is primarily determined by one of three workload dimensions—all engine-agnostic and extractable directly from request traces (Table 1):

(1) Per-request computation requirement (token lengths). The input prompt length determines prefill cost; the output length determines the number of decode iterations. Together, they set the baseline compute pressure per iteration and affect

knobs such as tensor-parallel-size.

For example, tensor-parallel-size (TP) partitions each model layer's weight across GPUs: every GPU computes a fraction of the computation, then synchronizes via all-reduce. When inputs are short, the computation is light and is memory-bandwidth-bound; and the fixed per-call latency of all-reduce dominates, making high TP wasteful [43]. As input length grows, the computation becomes the bottleneck, amortizing the fixed communication overhead—longer inputs favor higher TP.

(2) Inter-request KV$ hits. When requests share common prefixes, the engine can skip redundant prefill by reusing cached KV blocks, reducing compute cost while consuming GPU memory for the cache. The degree of reuse thus affects both compute and memory pressure and influences knobs such as KV$ block size and TP (they may share knobs).

For example, KV$ is allocated in fixed-size blocks, and block size sets the granularity of inter-request reuse. Larger blocks improve compute efficiency but may reduce reuse at prefix boundaries, since a block straddling two requests cannot be shared. The optimal size therefore depends on the workload's prefix-reuse pattern.

(3) Arrival dynamics. The rate and burstiness of incoming requests determine how the continuous-batching scheduler fills each iteration and affect knobs such as data-parallel-size, max-num-seqs, max-num-batched-tokens.

For example, max-num-seqs caps how many sequences the scheduler admits into one iteration batch. Too low, each batch underutilizes the GPU and throughput falls behind the arrival rate, causing queue buildup; too high, the scheduler waits to fill a large batch, adding queueing delay to early-arriving requests. Both extremes inflate end-to-end latency—the sweet spot shifts with arrival rate and burstiness.

These three dimensions are complementary: each captures an aspect of the workload that the other two cannot subsume. They are not fully independent—for example, bursty arrivals may shift the engine toward throughput-optimized settings that also interact with length-related knobs—but each dimension contributes unique information necessary to distinguish workloads that require different configurations.

4.2 The Workload Features and Similarity Metric

Having identified the three driving dimensions, we now define what constitutes a workload, extract a compact feature vector for each dimension, and build a similarity metric over it.

Workload definition. We define a workload W as a request trace collected from one serving cluster during a fixed time window (e.g., 1 day). Each request in W carries its (maybe anonymous) input prompt, output tokens, and arrival timestamp; from these fields we can extract all features needed to characterize the three dimensions above.

Our designed features for a workload. Given a workload, for each dimension we extract a sub-feature vector—L (Length), C (KV$ hits), A (Arrival dynamics)—as follows.

$$L(W) = \big[\log(1+\mathbb{E}[\ell]),\ \log\big(1+\tfrac{\mathrm{p95}(\ell)}{\mathbb{E}[\ell]+\epsilon}\big),\ \mathrm{cv}(\ell)\big]. \quad (1)$$

$$C(W) = \big[\log(1+\mathbb{E}[h]),\ \log\big(1+\tfrac{\mathrm{p95}(h)}{\mathbb{E}[h]+\epsilon}\big),\ \mathrm{cv}(h),\ \tfrac{\sum_i h_i}{\sum_i \ell_i+\epsilon}\big]. \quad (2)$$

$$A(W) = \big[\log(1+\lambda),\ \mathrm{cv}(\Delta t),\ \log(1+\mathrm{Fano}_{1\,\mathrm{s}})\big]. \quad (3)$$

In L, ℓ_i is the total token length of request i (input plus output tokens).[1] In C, h_i is the KV$ hit length of request i, computed assuming an idealized infinite cache to capture intrinsic prefix overlap rather than a specific engine's KV$ implementation. This is appropriate because serving systems actively size their KV$ to approach ideal reuse [26, 30], so the intrinsic overlap is engine-agnostic and enables cross-workload comparison. In A, λ is the average request arrival rate per GPU, and Δt denotes inter-arrival times.

Each sub-vector uses three types of statistics: $\mathbb{E}[\cdot]$ for central tendency, $\mathrm{cv}(\cdot)$ (coefficient of variation) for overall dispersion, and dimension-specific statistics described below. The ratio $\sum_i h_i/(\sum_i \ell_i + \epsilon)$ in C measures the overall KV$ hit rate. $\mathrm{Fano}_{1\,\mathrm{s}}$ in A is the Fano factor [32] (variance-to-mean ratio of per-second arrival counts), capturing burstiness that $\mathrm{cv}(\Delta t)$ alone cannot.

Why capture the tail? Beyond central tendency and overall dispersion, we include $\mathrm{p95}/\mathrm{mean}$ to quantify tail heaviness. In online LLM services, tail latency dominates user experience [34], and two workloads with identical mean and cv can have very different tails—leading to different optimal configurations for latency-sensitive knobs. The $\mathrm{p95}/\mathrm{mean}$ ratio isolates this tail behavior and is complementary to cv, which reflects spread but not its shape.

Why using the average request rate per-GPU? We normalize arrival rate by the number of GPUs rather than using the aggregate cluster-wide rate. In production, when the total request rate increases significantly, clusters scale out by adding GPU instances; conversely, they scale in when demand drops. The per-engine configuration, however, is determined by the load each engine sees—the per-GPU rate—not by how many engines the cluster happens to run. By normalizing, our feature vector becomes invariant to cluster size and total traffic volume, capturing only the workload characteristics that a per-engine tuner can act on. We further validate this in §5.1, where we show that the L-C-A profile remains stable as request volume fluctuates.

Why $\log(1+\cdot)$? The log transform makes distances reflect *relative* rather than *absolute* differences. Consider four workloads whose mean input lengths are 1 K, 2 K, 32 K, and 42 K tokens. Without the log, W_3–W_4 (10 K gap) appears

[1] For clusters with prefill-decode disaggregation [25, 40], we consider only input or output length in L for prefill and decode clusters, respectively.

ten times farther apart than W_1–W_2 (1 K gap), yet from the engine's perspective 1K → 2K (2× prefill cost) is far more significant than 32K → 42K (1.3×). The log converts multiplicative ratios into additive differences, making the metric scale-invariant.

Workload similarity metric. Given two workloads W and W', we concatenate L, C, and A into a single 10-dimensional feature vector for each, normalize each dimension with RobustScaler [27] to handle outliers, and define:

$$\mathrm{sim}(W, W') = \exp\big(-\|z(W) - z(W')\|_2\big),$$

where $z(W)$ is the normalized feature vector of W. The metric takes values in $(0, 1]$, with 1 indicating identical workloads. We use Euclidean distance because workload differences often arise from scale changes (e.g., input length) that cosine similarity would suppress.

5 Characterizing LLM Serving Configuration Tuning with Workload Similarity

5.1 Workload Temporal Locality Enables Configuration Reuse

We begin with the first open question raised in the introduction: *when does a tuned configuration remain valid?* Since the future workload is unknown at tuning time, practitioners must tune based on historical traces—but there is no a priori guarantee that a configuration optimized on the past remains good for the future. We answer this question empirically and then show that our workload similarity described in the previous section explains the result well.

Methodology. To investigate whether a configuration tuned on a past trace remains (near) optimal in the future, we use grid search to find the ground-truth best configuration for each day's trace using the methodology in §3. We then take Monday's best configuration and evaluate its peak sustainable request rate on each subsequent day (Tuesday through Sunday), comparing against that day's own grid-search oracle. Because an exhaustive grid search over 360 configurations across seven days is expensive, we limit this experiment to two representative workloads: chat and coder, serving Qwen3.5-27B and Qwen3-Coder-Next, respectively—the same models deployed on the clusters where the traces were collected, so the results are representative.

Same-cluster transfer succeeds. Figure 4 shows that within each cluster, Monday's best configuration achieves oracle-level peak sustainable request rate on all subsequent days, with a worst-case loss of 5.5%. The result holds for both the chat and coder clusters, indicating that the optimal configuration is stable across days within the same cluster.

Cross-cluster transfer fails. Figure 5 further shows the converse: configurations tuned on a different cluster's workload do not transfer. Specifically, applying the chat-optimal

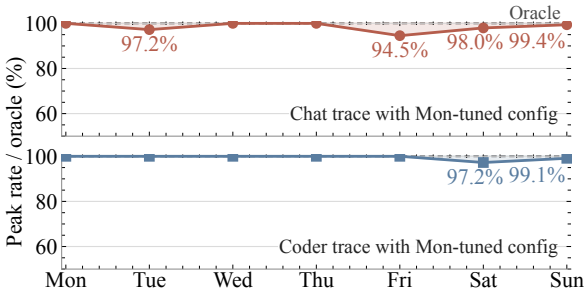

Figure 4: *Within each cluster (chat and coder), the configuration tuned on Monday achieves oracle-level peak sustainable request rate on all subsequent days, with a worst-case loss of 5.5%.*

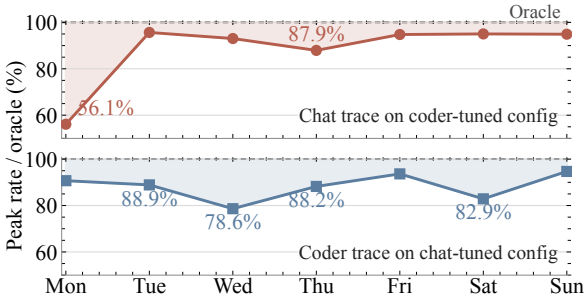

Figure 5: *Configurations tuned on one cluster's workload do not transfer to another: applying the chat-optimal configuration to coder (or vice versa) loses 5–44% peak sustainable request rate relative to each day's oracle.*

configuration to the coder workload—or vice versa—loses 5–44% in peak sustainable request rate relative to each day's oracle.

Workload temporal locality explains configuration reuse; our defined features capture it. We attribute the above two empirical observations to the workload temporal locality of LLM serving—the request mix within a cluster changing little from day to day. We validate on one full week; extending to longer horizons is ongoing.

While the tuning results in Figure 4 suggest workload temporal locality exists, it remains open when the workload will shift and thus retuning is needed, even with the same model and cluster. Fortunately, such locality can be captured with our similarity metric (§4): Figure 6 reports the pairwise overall similarity across seven days for three production clusters (chat, thinking, and coder). Within each cluster, the similarity score remains consistently high across all day pairs. Moreover, our ablation provided in §A.1 further confirms that the workload features relevant to configuration tuning—request lengths, KV$ hits, and arrival dynamics—change little from day to day. In contrast, cross-cluster similarity is consistently low, matching the cross-cluster transfer failure observed above.

Scale-stationary workload features. The observations above compare day-level aggregates; we now ask whether the L-C-A profile is preserved at finer time granularity. Figure 7 shows the per-10-minute L-C-A profile of each cluster over

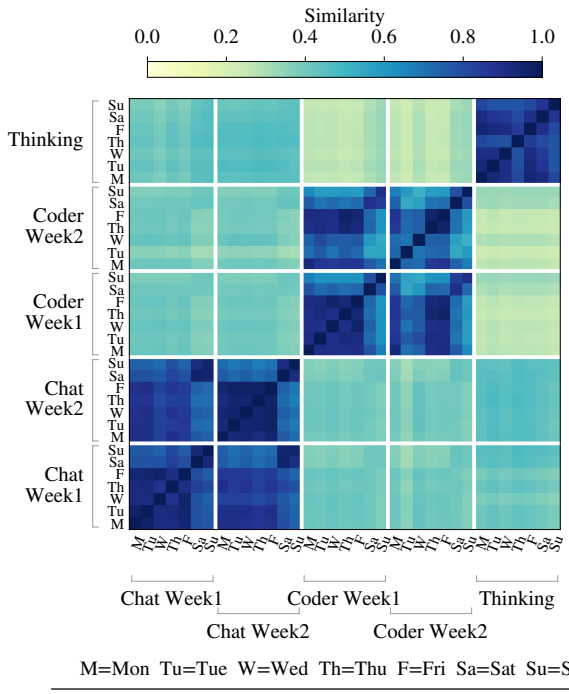

Figure 6: *For the same serving cluster, the workload exhibits locality across at least one week, while cross-workload similarity is low. Our similarity metric (§4.2) captures this well.*

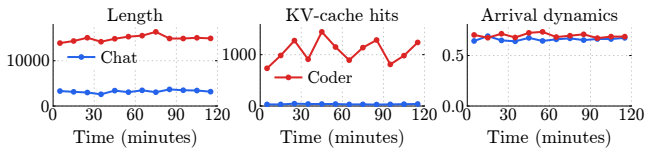

Figure 7: *The per-10-minute L-C-A profile of different workloads.*

a two-hour window. The profile remains remarkably stable across all buckets, even as the cluster-wide request rate fluctuates substantially (see Figure 2). This scale-stationarity has a practical implication: because the workload signature is independent of request volume, the tuner can evaluate configurations on a down-sampled trace at a fraction of the production rate and still obtain representative results.

> **Takeaway 1:** Within a single cluster, workloads exhibit temporal locality across a week, so a configuration tuned on one day's trace transfers to the following days. Our workload similarity metric captures this locality well in a scale-stationary way.

5.2 What Makes a Good Evaluation?

The previous section established *when* a tuned configuration remains valid; we now ask *how* to evaluate candidate configurations on a given workload trace so that the tuning result faithfully reflects the target workload's performance. This question is important because existing approaches differ widely in how they construct the workload used during evaluation:

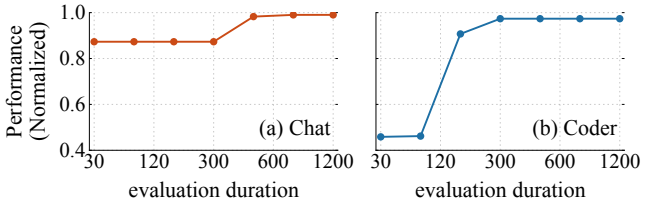

Figure 8: *An analysis of the best configuration tuned using different evaluation duration on (a) Chat and (b) Coder workloads.*

- **Fully synthetic:** draw requests from parametric distributions (e.g., random or fixed prompt lengths and Poisson arrivals) [37], and is the default for vLLM [4] and SGLang [3].
- **Semi-real:** evaluate with requests sampled from real prompts [12] but with synthetic arrival patterns, as some traces may miss the timestamp of requests. It preserves token-length statistics but not inter-request features like KV$ hits.
- **Real-trace replay:** replay (sampled) production traces directly, preserving all workload characteristics.

When a production trace is available, it is the obvious choice—but two questions arise: (1) how long must each evaluation run be, given that replaying a full 24-hour trace per candidate is infeasible? And (2) when no production trace exists, what properties must a synthetic workload preserve to yield a trustworthy evaluation? We answer both using the L-C-A similarity metric from §4.

Evaluation reliability stems from workload similarity, not wall-clock time. Given a production trace similar to the target workload (§5.1), we ask: how long must each evaluation replay be for the tuner to find a near-optimal configuration?

To determine how much trace is needed per evaluation, we conduct an experiment as follows: For each of our chat and coder clusters, we take one day's production trace and run an exhaustive grid search over all configurations, varying only the length of trace replayed per evaluation (from 30 s to 1200 s). We then compare the best configuration tuned with a given evaluation duration with the oracle configuration.

Figure 8 reports the result. Both workloads converge within a few minutes: chat reaches 98% of the oracle at 300 s, and coder reaches 97% at 180 s—far less than the full 24-hour trace. Note that the two workloads behave very differently at short evaluation windows: at 30 s, chat already achieves 87% of oracle performance while coder achieves only 46%. This can be explained via our similarity metric, which we describe next.

Our hypothesis is that, with insufficient evaluation duration, the workload features of the evaluated workload cannot yet approximate the real one. To validate this hypothesis, we compute the L-C-A similarity between each evaluation du-

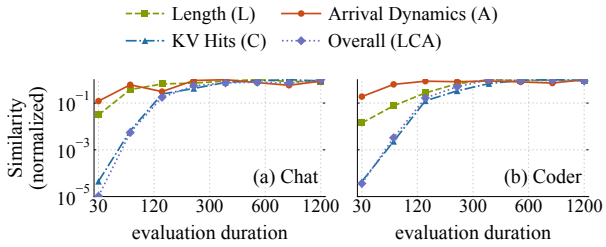

Figure 9: *An analysis of how the similarities between evaluated workload and real trace with different evaluation duration on (a) Chat and (b) Coder workloads.*

ration and the original sampled trace. As shown in Figure 9, the time at which overall similarity converges roughly aligns with the evaluation duration at which tuning finds a near-oracle configuration. This confirms our analysis on the two workloads.

An interesting phenomenon we find is that among the three LLM workload dimensions, KV$ reuse (C) is the slowest to converge for both workloads: at 30 s, both chat and coder have near-zero C similarity, because too few requests have been replayed for the KV$ hit distribution to stabilize. Length (L) and arrival dynamics (A), by contrast, converge faster. The reason is that it takes time for serving engines to warm their KV$. One may wonder why coder is more sensitive to KV$ than chat. The reason is that the KV$ reuse in chat workloads is much smaller than in coder (see Figure 2), so its performance impact is less.

> **Takeaway 2:** Evaluation reliability is governed by the similarity between the evaluated workload and the target workload in L-C-A feature space, not by wall-clock time alone. A few minutes of real-trace replay suffice for both workloads, but cache-dependent workloads are far more sensitive to short evaluation windows because the C dimension—which converges slowest—dominates their configuration landscape.

A faithful evaluator must preserve all workload features. Having established that similarity governs evaluation quality, we now use the same lens to diagnose *when* existing synthetic evaluators may fall short—and how to fix them. We repeat the grid-search methodology from the previous experiment, but now vary the evaluator type rather than the evaluation duration. We fix the per-evaluation time to 600 s (sufficient for convergence) and use three evaluators to tune a targeted day's trace (the second day in each cluster). For the real-trace replay, we use the first day's trace. We report each tuned configuration's peak sustainable rate relative to the per-day oracle.

Figure 10 shows a clear hierarchy. Real-trace tuning achieves 99–100% of the oracle across all three workloads. Fully synthetic evaluators lose 16% on chat and up to 35% on

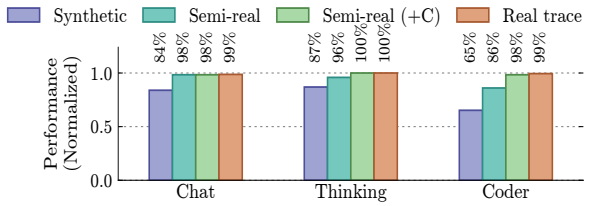

Figure 10: *An analysis of how the tuned performance with different evaluators on Chat, Thinking, and Coder workloads, respectively.*

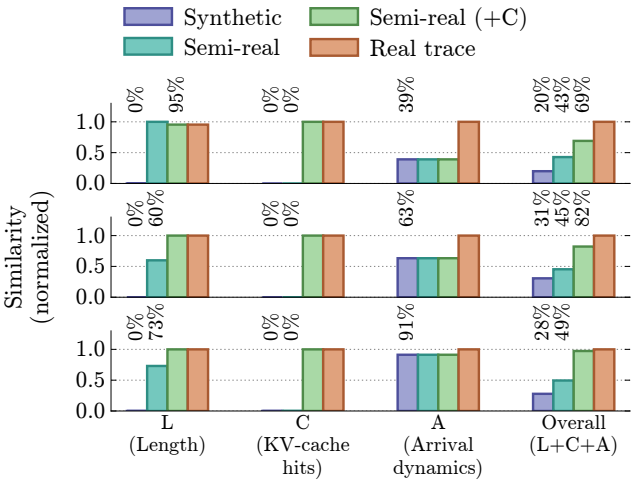

Figure 11: *An analysis of how the evaluated workload similarity differs from the real workload. The top, middle, and bottom panels correspond to Chat, Thinking, and Coder workloads, respectively.*

coder. Semi-real evaluators close much of the gap (98% on chat, 96% on thinking) but still lose 14% on coder, the most cache-dependent workload.

To understand *why* existing synthetic evaluators fall short, we decompose each evaluator's similarity to the production trace along the L-C-A dimensions, as shown in Figure 11. The pattern is clear and consistent across all workloads: arrival dynamics (A) are already comparable under synthetic traces—simple Poisson or empirical rate models capture this dimension adequately, which is as expected [2]. Moving from synthetic to semi-real recovers length similarity (L), because semi-real samples requests from the real trace. But the C dimension remains near zero for both synthetic and semi-real evaluators: neither generates the inter-request prefix overlaps that drive KV$ reuse in production. This missing dimension is precisely what causes the tuning gap on cache-heavy workloads like coder.

We therefore augment the semi-real evaluator by maintaining a sliding window that tracks the current KV$ hit rate of emitted requests; when sampling the next request, the sampler biases selection toward cache-hitting requests until the window's hit rate matches the target distribution, rather than drawing requests uniformly at random. The resulting Semi-real +C evaluator recovers the C dimension and lifts overall similarity to near real-trace levels. As Figure 10 confirms,

tuning with Semi-real +C reaches $\geq$98% of the oracle across all three workloads (98% on chat, 100% on thinking, 98% on coder)—within 2% of real-trace tuning—eliminating the need for a production trace.

Takeaway 3: Synthetic and semi-real evaluators fail because they miss inter-request cache dynamics (C dimension). Augmenting a semi-real evaluator with KV$ modeling (Semi-real +C) closes the gap to real-trace tuning, providing a practical path when production traces are unavailable.

5.3 Workload-Feature–to–Knob Correlations

In §5.1, we showed that applying chat's best configuration to coder loses up to 44% in peak sustainable rate. This section dives into the configuration differences and shows that our L-C-A features explain the direction of knob tuning.

Specifically, the best coder configuration is TP8 / MS 128 / MBT 32768 / LPT 32768, whereas chat prefers TP2 / MS 32 / MBT 16384 / LPT 20480 (each knob is explained below, and we have omitted other performance-unrelated changes). The two workloads differ systematically in two L-C-A dimensions: the tail heaviness of L and the KV$ hit rate C.

Tensor parallelism (TP): 2 $\rightarrow$ 8. TP partitions computation across GPUs (§4.1): higher TP speeds up individual requests but adds communication overhead. Coder's length distribution has a heavy tail (see Figure 2), and under colocated serving a single long prefill blocks every other request in the batch. TP8 cuts these long-tail prefills fast enough to prevent them from starving shorter requests; for chat, whose tail is much lighter, the overhead of TP8 outweighs the benefit. This motivates including the tail ratio in our feature design (§4.2).

Max sequences (MS): 32 $\rightarrow$ 128. MS caps the number of requests the scheduler admits into one batch of execution. Coder needs a larger MS because it has a much higher KV$ hit rate (see also Figure 2), so its prefill computation per request is much smaller than chat's. As a result, raising MS to 128 lets the engine exploit the higher parallelism that cache reuse creates.

Long-prefill threshold (LPT): 20480 $\rightarrow$ 32768. When a request's prefill exceeds LPT, the engine splits ("chunks") it into smaller pieces so that other requests can make progress between chunks. LPT trades off a long request's own speed against how much it blocks shorter requests—determined by both the tail of L (how many long requests exist) and C (how much cache reuse shrinks actual prefill work). Thus, it requires workload-specific tuning when these features change sharply.

```
[System Prompt]
You are an LLM inference systems engineer acting
as Researcher and Supervisor. Diagnose bottlenecks
from benchmark summaries and suggest better serving
engine configs. [...]
[Deployment State]
Env/hardware/model/workload summary
and latency target for the cluster.
[Recent Runs]
Latest runs: config, latency, tok/s, errors,
[... resource trends].
[Tested Configs]
Past config signatures; do not repeat.
[Harnesses] (see Figure 13)
[Tuning Rules]
Knob semantics, edit permissions, conservative
trust region.
[Your Response]
Return one JSON object: analysis, diagnosis,
knob_plan, suggested_configs (≤3), should_stop.
```

Figure 12: *Core structure of* AITUNER*'s prompt.*

6 Design and Implementation of AITUNER

Tuning environment and the design goal. AITUNER takes as input (1) a *target workload trace*—a time-stamped request sequence collected from a production cluster or synthetic—but we favor those from the production cluster and (2) a *execution environment*: the model to tune, the GPU the model runs on, the engine used and the SLO target. The goal is to find a high-performing configuration for the target workload while consuming as few GPU-hours as possible, since each trial requires deploying the model on the GPU testbed and replaying the trace. AITUNER achieves this by running on a CPU server with an agentic tuning loop that automates the evaluate–search loop described in the introduction with LLM. LLMs are a natural fit: they can automate the trial loop and reason about the causal relationships between configuration knobs and performance.

The agentic loop. Specifically, given an initial configuration, the loop first evaluates the configuration and collects the results, which are fed back to the LLM to generate a new configuration; the iteration continues until the search budget is exhausted. The simplified prompt of the LLM is shown in Figure 12. Note that the prompt omits the details of some basic harness, i.e., the doc of the possible tuned configuration knobs to avoid generating invalid configurations, and a sliding window to feed the model the history of tuning without reaching the model context limit.

Our tuner uses OpenAI API-compatible interfaces [22] thus it can work with any LLM, and our current choice is GPT-5.4. We use a frontier model because stronger reasoning reduces the number of GPU trials needed, and the API cost is negligible compared to the GPU-hours saved (§7). Moreover, we don't use tools to filter out invalid configurations because we found it is extremely rare for a powerful model to generate invalid configurations.

Harness: Adjusting TP

Procedure: (1) On the active environment, probe adjacent TP choices in the current effective-prefill regime. (2) From the probes, estimate the local prefill service-latency slope, queue-delay growth, and the high-TP queueing knee. (3) For the live window, compute real_prefill_per_req, prefix_hit, burst_prefill_load_per_gpu, active replica count, and TTFT-SLO gap; project each TP choice onto the same boundary. (4) If TP↑ lowers prefill latency and the projected point stays left of the high-TP queueing knee, open the higher-TP branch. If concurrency headroom shrinks too much, keep moderate TP; if prefill is no longer the active bottleneck, switch branches.

Figure 13: *An example of dedicated TP-diagnosis harness injected into system prompt shown in Figure 12 for LLM agent to tune.*

Two retrofitted designs for tuning efficiency and effectiveness. A key design goal of AITUNER is to find a near-optimal configuration within a tight GPU-hour budget. Based on our characterizations in §5, we apply two techniques that directly reduce GPU time per tuning session:

First, §5.2 showed that evaluation reliability depends on the L-C-A similarity between the replayed workload and the target trace, not on wall-clock time. AITUNER exploits this by monitoring the real-time L-C-A feature vector during each trial and comparing it against the target trace's feature vector. Once all three dimensions have converged, the evaluation terminates early; if the C dimension remains unconverged—typically because the KV$ has not yet warmed—the evaluation continues. This saves GPU time on fast-converging workloads (e.g., chat) while avoiding premature termination on cache-dependent ones (e.g., coder). Unlike SCOOT [12], which uses a fixed 30-minute evaluation window per trial, AITUNER's adaptive stopping reduces per-trial evaluation time by $\sim$70% on average.

Second, while adaptive stopping reduces per-trial time, the *number* of trials remains the dominant cost when the agent lacks domain knowledge. Without guidance, the basic tuner often spends trials exploring irrelevant knob families before converging on the right one, and may exhaust the search budget before reaching a near-optimal configuration (§7.3). To cut this wasted exploration, we encode domain knowledge about workload-feature-to-knob correlations (§5.3) as *harnesses* injected into the agent's prompt. Each harness is a rule that maps a bottleneck regime—identified by the current metrics (e.g., high TTFT, low GPU utilization, low queueing delay)—to the knob family most likely to resolve it, together with a guard condition that prevents harmful side effects. Figure 13 shows a concrete example of how to tune the TP knob. Currently, we provide a harness for each of the knobs, and the harness can easily evolve as the engine evolves by adding new harnesses.

7 Evaluation of AITUNER

7.1 Evaluation setup

Evaluated workloads and engine used. We evaluate on five production serving clusters at COMPANYX, spanning two model scales, two GPU platforms, and three different applications:

- *(Chat) Qwen3.5-27B on H20* — it uses H20 to serve a Chat application serving both prefill and decode workloads, a.k.a., PD colocation.
- *(Chat) Qwen3.5-27B on RTX 5090* — the application is the same as the above except that the cluster uses a different GPU (5090).
- *(Thinking-prefill) Qwen3-235B-A22B on H20* — it is the cluster used for serving thinking requests, and it is deployed under prefill-decode disaggregation [40].
- *(Thinking-decode) Qwen3-235B-A22B on H20* — it serves the decode requests in companion with the above cluster.
- *(Coding) Qwen3-Coder-Next on H20* — it serves coding agents like Claude Code [9] and the deployment is PD colocation.

All experiments tune and evaluate directly on COMPANYX's internal serving engine, and we set the SLO targets to match those of the production clusters. Due to confidentiality requirements, we omit the SLO details.

Baselines. We compare AITUNER against the following:

- *Deployed configuration.* The configuration currently running in each production cluster, hand-tuned by COMPANYX engineers. Some clusters have been carefully tuned; others have received limited attention due to time constraints.
- *Community default.* Some of our models are open-source and so there are community configurations from open-source engines like vLLM [6]. We mimic engine-agnostic configurations from vLLM and use them as an under-tuned starting configuration.
- *SCOOT* [12]. The state-of-the-art automatic tuner for LLM serving engines: SCOOT treats the configuration space as a black box and uses Gaussian-process surrogate models to select the next candidate. We have ported the open-source SCOOT code [1] to tune on our internal engines. SCOOT failed to find any valid configuration on the Qwen3-Coder-Next cluster within its iteration budget, so we omit its results on that cluster.
- *Naive agentic tuner.* A vanilla LLM-based tuning loop using the same model as our tuner but without the harnesses described in §6.

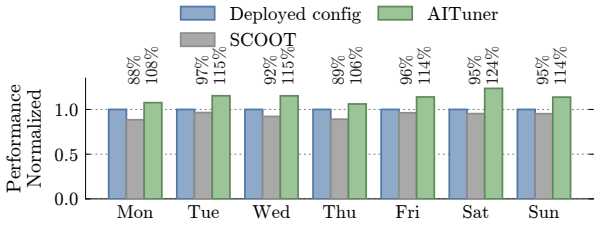

Figure 14: *Tuned results on (Chat) Qwen3.5-27B on H20.*

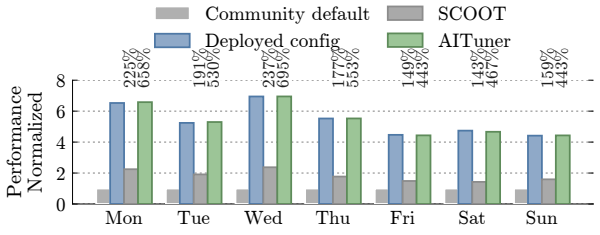

Figure 15: *Tuned results on (Chat) Qwen3.5-27B on RTX 5090.*

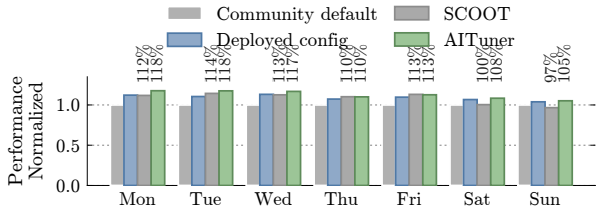

Figure 16: *Tuned results on (Thinking-prefill) Qwen3-235B-A22B on H20.*

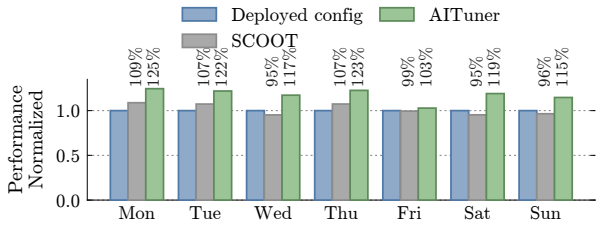

Figure 17: *Tuned results on (Thinking-decode) Qwen3-235B-A22B on H20.*

For all automated tuners (SCOOT, naive agentic tuner, and AITUNER), each trial uses AITUNER's adaptive monitor (§6) for early stopping; without it, tuning time would be prohibitively long.

7.2 Effectiveness of Tuning

Methodology. For each cluster, we take the previous Sunday's production trace as the tuning input and run each automated tuner with a budget of 10 iterations to produce the best configuration found during the search (§7.3). We then freeze the best configuration and evaluate it on every day of the following week (Monday through Sunday), reporting peak sustainable rate—the maximum per-GPU request rate at which SLO attainment remains above 99% (§3). This setup mirrors the temporal-locality experiment in §5.1. For each day, we normalize peak sustainable rate to the deployed configuration's rate, so 1.0 means matching the hand-tuned production setting.

Overall results. Figures 14–19 show the normalized peak sustainable rate of each method across all five clusters over one week. AITUNER consistently discovers configurations that match or outperform the deployed configuration on every cluster and every day.

On Qwen3.5-27B/H20 (Figure 14), AITUNER improves over the deployed configuration by 6–24% (14% on average), while SCOOT underperforms the deployed baseline on all seven days. On Qwen3-235B-A22B decode (Figure 17), the gain is even larger: 3–25% (18% on average), because the decode cluster had received limited manual tuning. On Qwen3-Coder-Next (Figure 19), AITUNER improves by 3–21% (13% on average). On Qwen3-235B-A22B prefill (Figure 16), the gain is more modest at 1–6%, as this cluster was already reasonably well tuned. In contrast, on Qwen3.5-27B/5090 (Figure 15), AITUNER matches the deployed configuration within 2%—this cluster has been carefully hand-tuned by COMPANYX engineers and leaves little headroom. Even in this case, both AITUNER and the deployed configuration outperform the community default by 4–7×, confirming that per-cluster tuning is essential.

Across all five clusters, AITUNER improves per-GPU performance by 9.5% on average over the deployed configuration, with three out of five clusters seeing improvements of 13% or more.

Why SCOOT struggles. SCOOT fails to improve over the deployed baseline on any of the clusters where we evaluate it. After inspecting its search process, we found SCOOT repeatedly proposes infeasible configurations that fail before serving starts, primarily due to startup OOM. The root cause is that its Bayesian search space is not deployment-aware—it neither considers the hardware memory envelope nor the runtime invariants of our engine version. As a result, SCOOT spends most of its budget on invalid configurations, a failure mode that AITUNER avoids by leveraging the LLM's understanding of engine semantics to prune the search space (§6).

7.3 Efficiency of Tuning

Methodology. We measure tuning efficiency by the number of *tuning iterations* required to converge to the final configuration. Each iteration evaluates a candidate configuration at multiple offered-load levels (i.e., different trace sampling rates) to determine its peak sustainable rate, so one iteration corresponds to multiple benchmark runs. Because the evaluation time per iteration (about one hour) dominates the LLM API call time (seconds per invocation), iteration count is a faithful proxy for wall-clock tuning cost.

For all tuning, we use the same iteration budget of 10 for all tuners. For reference, an exhaustive grid search over the same

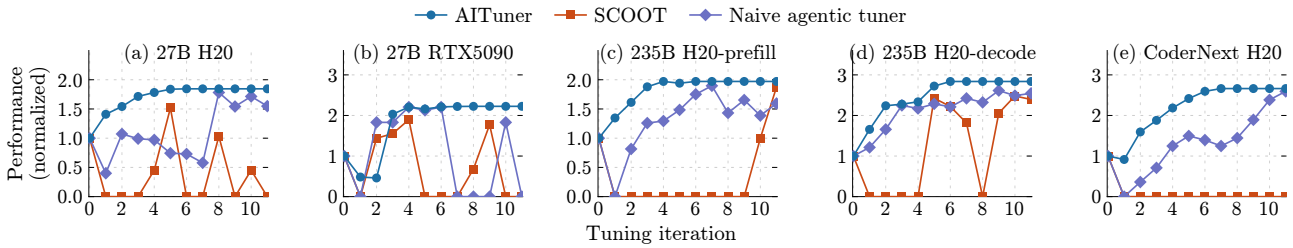

Figure 18: *Tuning iterations across five clusters: (a) Qwen3.5-27B on H20, (b) Qwen3.5-27B on RTX 5090, (c) Qwen3-235B-A22B prefill-only on H20, (d) Qwen3-235B-A22B decode-only on H20, and (e) Qwen3-Coder-Next on H20.*

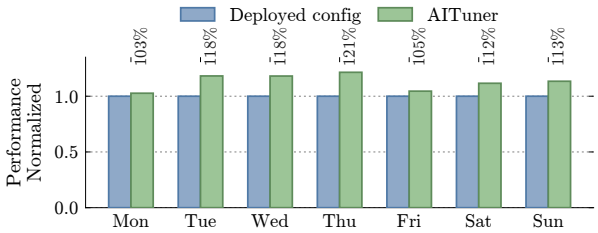

Figure 19: *Tuned results on (Coding) Qwen3-Coder-Next on H20.*

knob space requires at least 360 iterations (§3), so even a budget of 10 represents a two-order-of-magnitude reduction.

Overall results. Figure 18 plots the best peak sustainable rate found so far as a function of the tuning iteration for each of the five clusters. All values are normalized to the deployed configuration, and the initial configuration is the community default. AITUNER converges to its final configuration within 3–6 iterations across all five clusters (average 4.6), after which performance remains flat. In contrast, the naive agentic tuner and SCOOT both fail to converge within the same budget, but for different reasons.

The naive agentic tuner occasionally discovers configurations comparable to AITUNER's (e.g., iteration 5 in Figure 18a and 6 in d), but it lacks the ability to recognize a good configuration and stop. Instead, it continues searching and repeatedly regresses—in Figure 18a, performance drops from $\sim$1.5$\times$ at iteration 5 to $\sim$0.5$\times$ by iteration 8. This oscillation is a direct consequence of the missing harnesses described in §6: without convergence detection, the tuner treats every iteration as a fresh exploration step. Interestingly, we observe that a powerful frontier model (GPT-5.4) can discover near-optimal configurations even without harnesses, suggesting that stronger models may eventually reduce the need for structured guidance.

SCOOT suffers from a different failure mode: it spends most of its budget on infeasible configurations that crash before serving a single request. Across the five clusters, $\sim$80% of SCOOT's iterations produce zero throughput, leaving too few valid observations for its surrogate model to learn from.

8 Related Work

LLM inference serving engine optimization. Optimizing the performance of LLM serving engines is critical for reducing cost and meeting latency targets at scale [7, 40, 28, 19, 41, 17]. These optimizations are typically workload-driven, and thus each one may introduce configuration knobs that must be tuned—whether to enable a feature, and if so, with what parameters (e.g., the chunk size in Sarathi-Serve's chunked prefill [7]). This creates a growing, workload-dependent configuration space that demands a principled automatic tuning approach. We systematically analyze how to tune these knobs both accurately and efficiently.

Workload characterization and evaluation for LLM serving. Several studies characterize production LLM serving workloads [31, 36, 30], revealing important phenomena such as bursty arrivals, skewed output lengths, and workload-dependent KV-cache reuse. However, these results remain high-level observations: they do not directly tell a practitioner *how* to translate a given workload profile into an effective engine configuration. We bridge this gap by defining a workload similarity metric that directly captures the features determining serving performance.

LLM-assisted systems optimization. We continue the line of research on using LLMs to drive systems optimization [21, 10, 11]. These works target general-purpose optimization and largely treat the problem domain as a black box. We open the black box for LLM serving: we define a workload similarity metric grounded in serving-specific observations to guide tuning effectiveness, and encode domain knowledge in workload feature vectors to accelerate the search.

9 Conclusion

This paper shows that a compact workload similarity metric can bridge the gap between offline tuning and online serving for LLM serving configurations: it tells *when* to retune and *how* to evaluate candidates, and its feature-to-knob correlations compress the search space. AITUNER builds on this metric, using an LLM to translate the correlations into pruning rules for efficient search. On five production clusters, AITUNER improves per-GPU performance by 9.5% on average over production baselines—with three clusters seeing 13%+ gains—with a shorter tuning time.

References

[1] Scoot. https://github.com/Ketonmi/SCOOT, 2025.

[2] Poisson distribution. https://en.wikipedia.org/wiki/Poisson_distribution, 2026. Wikipedia.

[3] Sglang benchmarks: bench_serving.py. https://github.com/sgl-project/sglang/blob/main/python/sglang/bench_serving.py, 2026.

[4] vllm benchmarks: serve.py. https://github.com/vllm-project/vllm/blob/main/vllm/benchmarks/serve.py, 2026.

[5] vllm PR #34936. https://github.com/vllm-project/vllm/pull/34936, 2026.

[6] vllm releases. https://github.com/vllm-project/vllm/releases, 2026.

[7] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 117–134. USENIX Association, 2024.

[8] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. SARATHI: efficient LLM inference by piggybacking decodes with chunked prefills. *CoRR*, abs/2308.16369, 2023.

[9] Anthropic. Claude code overview. https://code.claude.com/docs/en/overview, 2026.

[10] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive LLM driven zeroth-order optimization. *CoRR*, abs/2602.20133, 2026.

[11] Audrey Cheng, Shu Liu, Melissa Z. Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya A. Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How AI is upending systems research. *CoRR*, abs/2510.06189, 2025.

[12] Ke Cheng, Zhi Wang, Wen Hu, Tiannuo Yang, Jianguo Li, and Sheng Zhang. SCOOT: slo-oriented performance tuning for LLM inference engines. In Guodong Long, Michale Blumestein, Yi Chang, Liane Lewin-Eytan, Zi Helen Huang, and Elad Yom-Tov, editors, *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*, pages 829–839. ACM, 2025.

[13] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[14] Hugging Face. Hugging face. https://huggingface.co, 2026.

[15] GitHub. Accelerate your development speed with copilot. https://copilot.github.com, 2026.

[16] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 443–462. USENIX Association, 2020.

[17] Jinwoo Jeong and Jeongseob Ahn. Accelerating LLM serving for multi-turn dialogues with efficient resource management. In Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, pages 1–15. ACM, 2025.

[18] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[19] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. Infinigen: Efficient generative inference of large language models with dynamic KV cache management. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 155–172. USENIX Association, 2024.

[20] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 58. ACM, 2021.

[21] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025.

[22] OpenAI. Build on the openai api platform. https://platform.openai.com/docs/overview, 2026.

[23] OpenAI. Chatgpt. https://chatgpt.com, 2026.

[24] OpenRouter. The unified interface for llms. https://openrouter.ai, 2026.

[25] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, pages 118–132. IEEE, 2024.

[26] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation - A kvcache-centric architecture for serving LLM chatbot. In Haryadi S. Gunawi and Vasily Tarasov, editors, *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, pages 155–170. USENIX Association, 2025.

[27] scikit-learn developers. Robustscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html, 2024.

[28] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 173–191. USENIX Association, 2024.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[30] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. Kvcache cache in the wild: Characterizing and optimizing kvcache cache at a large cloud provider. In Deniz Altinbüken and Ryan Stutsman, editors, *Proceedings of the 2025 USENIX Annual Technical Conference, USENIX ATC 2025, Boston, MA, USA, July 7-9, 2025*, pages 465–482. USENIX Association, 2025.

[31] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize LLM serving systems. In Luiza Antonie, Jian Pei, Xiaohui Yu, Flavio Chierichetti, Hady W. Lauw, Yizhou Sun, and Srinivasan Parthasarathy, editors, *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, V.2, KDD 2025, Toronto ON, Canada, August 3-7, 2025*, pages 5831–5841. ACM, 2025.

[32] Wikipedia. Fano factor. https://en.wikipedia.org/wiki/Fano_factor, 2026.

[33] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 640–654. ACM, 2024.

[34] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *CoRR*, abs/2305.05920, 2023.

[35] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin,

and Jingren Zhou. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs, editors, *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, pages 1030–1045. ACM, 2025.

[36] Yuxing Xiang, Xue Li, Kun Qian, Wenyuan Yu, Ennan Zhai, and Xin Jin. Servegen: Workload characterization and generation of large language model serving in production. *CoRR*, abs/2505.09999, 2025.

[37] Tianhao Xu, Yiming Liu, Xianglong Lu, Yijia Zhao, Xuting Zhou, Aichen Feng, Yiyi Chen, Yi Shen, Qin Zhou, Xumeng Chen, Ilya Sherstyuk, Haorui Li, Rishi Thakkar, Ben Hamm, Yuanzhe Li, Xue Huang, Wenpeng Wu, Anish Shanbhag, Harry Kim, Chuan Chen, and Junjie Lai. Aiconfigurator: Lightning-fast configuration optimization for multi-framework LLM serving. *CoRR*, abs/2601.06288, 2026.

[38] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 521–538. USENIX Association, 2022.

[39] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Blitzscale: Fast and live large model autoscaling with O(1) host caching. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 275–293. USENIX Association, 2025.

[40] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 193–210. USENIX Association, 2024.

[41] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Nanoflow: Towards optimal large language model serving throughput. In Lidong Zhou and Yuanyuan Zhou, editors, *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, pages 749–765. USENIX Association, 2025.

[42] Kan Zhu, Haiyang Shi, Le Xu, Jiaxin Shan, Arvind Krishnamurthy, Baris Kasikci, and Liguang Xie. Polyserve: Efficient multi-slo serving at scale. *CoRR*, abs/2507.17769, 2025.

[43] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. Megascale-infer: Efficient mixture-of-experts model serving with disaggregated expert parallelism. In Marília Curado, Christian Esteve Rothenberg, George Porter, and Srikanth Kandula, editors, *Proceedings of the ACM SIGCOMM 2025 Conference, SIGCOMM 2025, São Francisco Convent, Coimbra, Portugal, September 8-11, 2025*, pages 592–608. ACM, 2025.

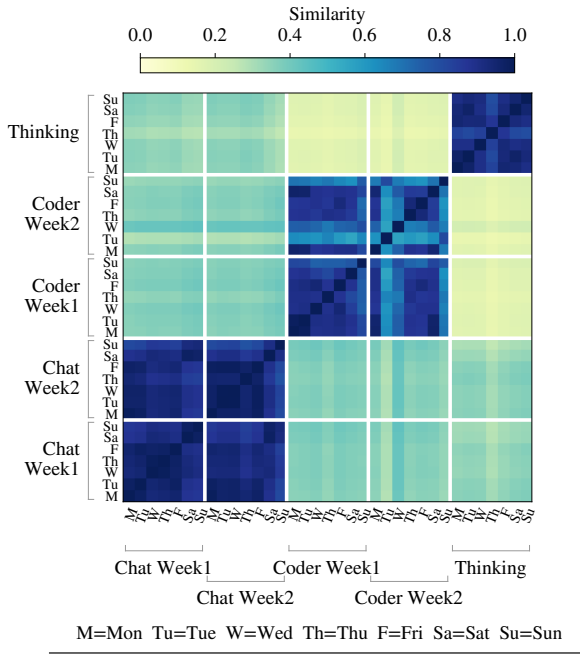

Figure 20: *Workload similarity considering only L (Length).*

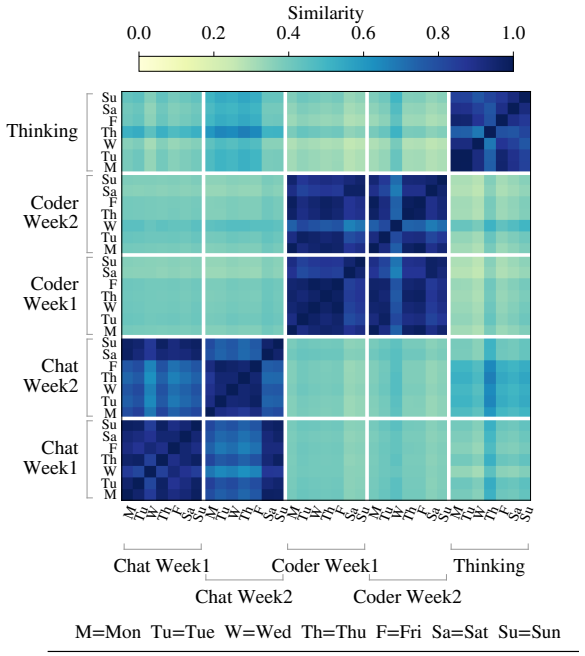

Figure 21: *Workload similarity considering only C (KV$ reuse).*

A Appendix

A.1 Workload similarity by feature family

Figures 20, 21, and 22 decompose weekday workload similarity by the three feature families we proposed in §4.

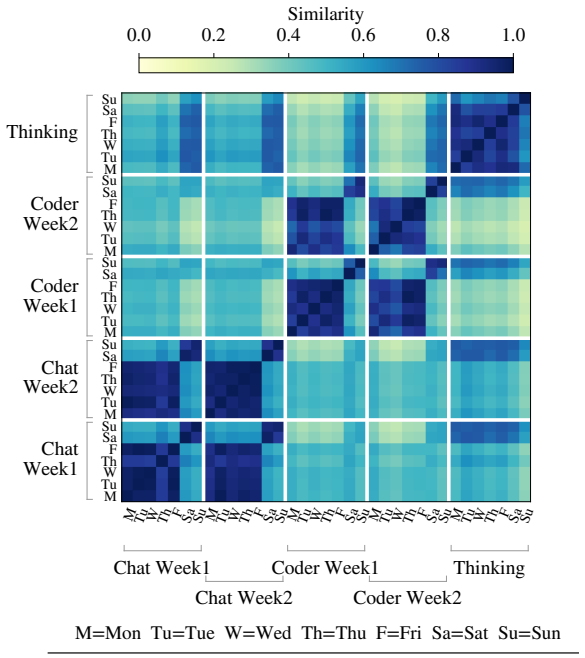

Figure 22: *Workload similarity considering only A (Arrival pattern).*