MIXTUNER: DATABASE–KERNEL CO-TUNING VIA CONTEXT TRANSFER AND STAGED SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Automatic tuning improves database performance by replacing manual trial and error with configuration search and workload measurements. Database settings govern resource use, while operating-system kernel settings determine the mechanisms available to support it. Joint tuning can align database resource use with kernel capabilities that a single-layer search leaves fixed. Selecting these settings together requires connecting differently organized configuration knowledge and managing the cost of evaluating configuration pairs. We introduce MixTuner, a database–kernel co-tuning framework based on context transfer and staged search. A large language model first selects database knobs and suggests numerical ranges from configuration documentation. The selected knob information and functional groups then guide Linux Kconfig menu filtering and option recommendations. After the resulting kernel is built and deployed, coarse-to-fine Bayesian optimization selects database values through measurements on that fixed kernel. The grouped context connects the configuration hierarchies, while the staged schedule shares kernel preparation across repeated database trials. Across 36 MySQL and PostgreSQL TPC-C and TPC-H scenarios, MixTuner improves over the officially recommended configurations in 35. On MySQL TPC-H, MixTuner achieves lower P95 latency than the state-of-the-art database-only and kernel-only baselines.

1 INTRODUCTION

Database systems store, organize, and retrieve data to support transaction processing and analytical applications. Their performance depends on how database settings and operating-system kernel settings use the same hardware. For a fixed machine, higher transaction throughput means serving more requests per second, while lower query latency means shorter waits for analytical results. Configuration tuning can improve both without changing the workload or adding hardware. Experts typically tune two layers: at the **database layer**, they select values for database parameters, commonly called *knobs*, that control buffering, concurrent execution, and logging; at the **kernel layer**, they select Linux build options that determine which memory-management, scheduling, and I/O mechanisms support these operations (The Linux Kernel Contributors, 2026). However, manually tuning both layers involves repeated configuration changes and workload runs, which are time-consuming and require domain knowledge of database internals and kernel behavior.

Recently, automatic tuners have reduced manual effort by selecting parameters and using measurements to guide search. **Database tuners** rank knobs from observations (Van Aken et al., 2017; 2021), search low-dimensional representations (Kanellis et al., 2022), or stop unpromising evaluations early (Yue et al., 2026). Knowledge-guided methods use documentation, large language models (LLMs), or DBMS source code to select knobs and guide their values (Lao et al., 2024; Yan et al., 2025; Huang et al., 2025; Zhang et al., 2026). **Kernel tuners** search kernel configurations using learned performance models (Jung et al., 2026; Sun et al., 2026), configuration hierarchies (Chen et al., 2024), domain knowledge (Lin et al., 2026b), or learned configuration policies (Lin et al., 2026a). However, these single-layer approaches leave the parameter relationships and mutual constraints between database and kernel configurations outside their search, potentially resulting in suboptimal database performance.

For example, if native Asynchronous Input/Output (AIO) support is disabled through Linux's `CONFIG_AIO` build option, MySQL's InnoDB cannot use that kernel path for data-page reads and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

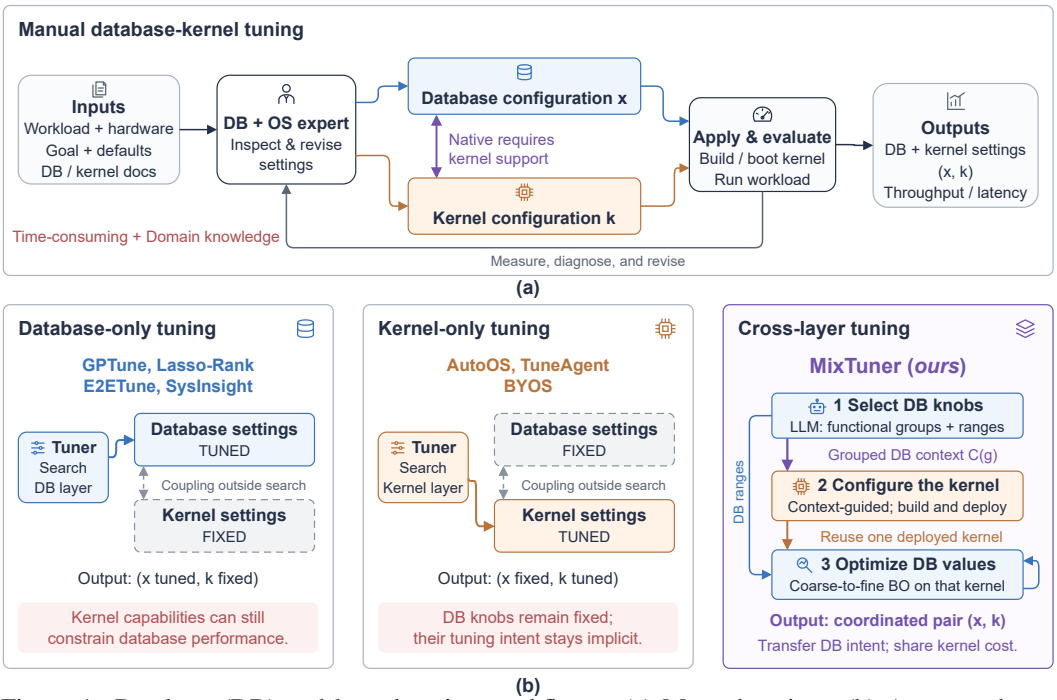


Figure 1: Database (DB) and kernel tuning workflows. (a) Manual tuning. (b) Automated tuning: database-only (left) and kernel-only (middle) methods optimize. MixTuner (right) combines database and kernel domain knowledge to coordinate tuning across both layers.

writes, even when `innodb_use_native_aio` requests it (The Linux Kernel Contributors, 2024; Oracle, 2026). Similarly, PostgreSQL's explicit huge-page allocation requires the kernel build options `CONFIG_HUGETLBFS=y` and `CONFIG_HUGETLB_PAGE=y`, together with sufficient default-size huge pages reserved through the runtime setting `vm.nr_hugepages` (The PostgreSQL Global Development Group, 2026a). These examples motivate joint, cross-layer tuning of database and kernel configurations instead of treating them as isolated processes. Figure 1(a) illustrates the manual workflow, while Figure 1(b) compares database-only, kernel-only, and our cross-layer tuning. We therefore pose the research question: *How can configuration knowledge connect the two layers while keeping the search affordable?* Two challenges shape this problem:

Knowledge misalignment between database and kernel. The first challenge is the lack of correspondence between database and kernel configuration knowledge, which makes it difficult to identify the kernel mechanisms relevant to the database settings being tuned. Database documentation groups knobs by functions such as buffering and logging; Linux Kconfig organizes options by subsystem and dependency (The Linux Kernel Contributors, 2026). For example, selecting PostgreSQL's `shared_buffers` and `work_mem` expresses an intention to tune shared buffering and query working memory. A kernel tuner needs that functional context to inspect relevant memory-management options. Giving two tuners the same throughput or latency goal leaves this database-specific intent implicit.

Combinatorial explosion of the joint search space. The second challenge is the size of the joint configuration space: every database configuration can be paired with many feasible kernel configurations. For illustration, ten database knobs with five values each and twenty binary kernel options yield $5^{10}2^{20} = 1.024 \times 10^{13}$ raw pairs before dependency filtering. Exploring these pairs also incurs unequal evaluation costs. A database trial requires applying knob values and running the workload, whereas changing a kernel build additionally requires compilation, boot validation, and deployment. This combinatorial growth, compounded by expensive kernel transitions, makes broad joint exploration costly. An affordable search must reduce the candidate space and limit repeated kernel preparation.

We introduce MixTuner, which addresses these challenges through *context transfer* and *staged search*. *Context transfer* (addressing Challenge 1) uses a grouped intermediate representation containing selected knob names, functional descriptions, and documented relationships. The LLM constructs this representation during database knob selection; the kernel tuner uses it both to filter subsystem menus and to recommend option values. This interface connects database and kernel knowledge without requiring a predefined one-to-one mapping between knobs and kernel symbols. The selected knobs also define reduced ranges for database search. *Staged search* (addressing Challenge 2) then proceeds in two steps: kernel tuning generates and deploys a kernel conditioned on the selected-knob context, and database tuning uses coarse-to-fine Bayesian optimization to search knob values on that fixed kernel. Many database trials therefore share one deployment, and their measurements guide subsequent proposals within the reduced search space.

Our contributions are threefold:

- We formulate database–kernel co-tuning as selecting a configuration pair for a shared workload and fixed runtime environment. The formulation relates single-layer search to the joint space and identifies cross-layer knowledge misalignment and combinatorial search as its two central challenges.
- We develop MixTuner, a co-tuning framework that combines context transfer with staged search. Grouped database context guides kernel-menu filtering and option assignment, while the staged procedure reuses the resulting kernel for coarse-to-fine database value optimization.
- We evaluate MySQL and PostgreSQL on TPC-C and TPC-H across 36 scale–concurrency scenarios. MixTuner improves throughput or P95 request latency in 35 scenarios over the officially recommended configurations. On MySQL TPC-H, MixTuner also achieves lower P95 latency than the state-of-the-art database-only and kernel-only baselines.

2 Problem Formulation

Let w denote a fixed workload and h the hardware and fixed runtime environment. A database configuration is a vector of knob values $\mathbf{x} \in \mathcal{X}_{\mathrm{db}} = \prod_{i=1}^{n} \mathcal{X}_i$, where each $\mathcal{X}_i$ is a numerical or categorical domain. These knobs control database behavior such as buffering, concurrency, and logging. A kernel configuration assigns values to Linux Kconfig options and is denoted by $\mathbf{k} \in \mathcal{K}(h)$, where $\mathcal{K}(h)$ contains build configurations that satisfy the architecture and Kconfig language constraints. Thus, $\mathbf{k}$ specifies build-time kernel options controlling the available memory-management, scheduling, and I/O mechanisms; runtime operating-system settings are held fixed in h. Let $\mathcal{F}(w,h) \subseteq \mathcal{X}_{\mathrm{db}} \times \mathcal{K}(h)$ denote the configuration pairs that successfully execute workload w in environment h. The reference settings $(\mathbf{x}^0, \mathbf{k}^0)$ are the officially recommended configurations, instantiated by the vendor-provided database settings and the operating-system distribution's kernel configuration in use before tuning begins. We formalize the search spaces for the three tuning tasks as follows:

$$\begin{aligned} \text{Database-only tuning:} \quad & \mathcal{S}_{\mathrm{db}}(\mathbf{k}^0) = \mathcal{F}(w,h) \cap \big(\mathcal{X}_{\mathrm{db}} \times \{\mathbf{k}^0\}\big), \\ \text{Kernel-only tuning:} \quad & \mathcal{S}_{\mathrm{ker}}(\mathbf{x}^0) = \mathcal{F}(w,h) \cap \big(\{\mathbf{x}^0\} \times \mathcal{K}(h)\big), \\ \text{Co-tuning:} \quad & \mathcal{S}_{\mathrm{co}} = \mathcal{F}(w,h) \cap \big(\mathcal{X}_{\mathrm{db}} \times \mathcal{K}(h)\big). \end{aligned} \tag{1}$$

Database-only tuning fixes the kernel, and kernel-only tuning fixes the database. Co-tuning selects both configurations. Let $J(\mathbf{x}, \mathbf{k}; w, h)$ be the workload performance objective, with larger values indicating better performance, for example, transaction throughput or negative 95th-percentile latency.

Configuration knowledge describes what settings mean and which values they allow. We represent database knowledge by $\mathcal{T}_{\mathrm{db}}$: documented knob names, functional descriptions, types, defaults, and legal values, organized into functional groups. Kernel knowledge $\mathcal{T}_{\mathrm{ker}}$ comprises Kconfig menus, option symbols, help text, types, and dependency and visibility rules. Using this knowledge, the optimization target for a search space $\mathcal{S}$ is a configuration pair $(\mathbf{x}^\star, \mathbf{k}^\star)$ that maximizes workload performance:

$$(\mathbf{x}^\star, \mathbf{k}^\star) = \underset{(\mathbf{x},\mathbf{k}) \in \mathcal{S}}{\arg\max} J(\mathbf{x}, \mathbf{k}; w, h), \quad \mathcal{S} \in \big\{\mathcal{S}_{\mathrm{db}}(\mathbf{k}^0), \mathcal{S}_{\mathrm{ker}}(\mathbf{x}^0), \mathcal{S}_{\mathrm{co}}\big\}. \tag{2}$$

Section 3 introduces MixTuner, our method for database–kernel co-tuning in $\mathcal{S}_{\mathrm{co}}$.

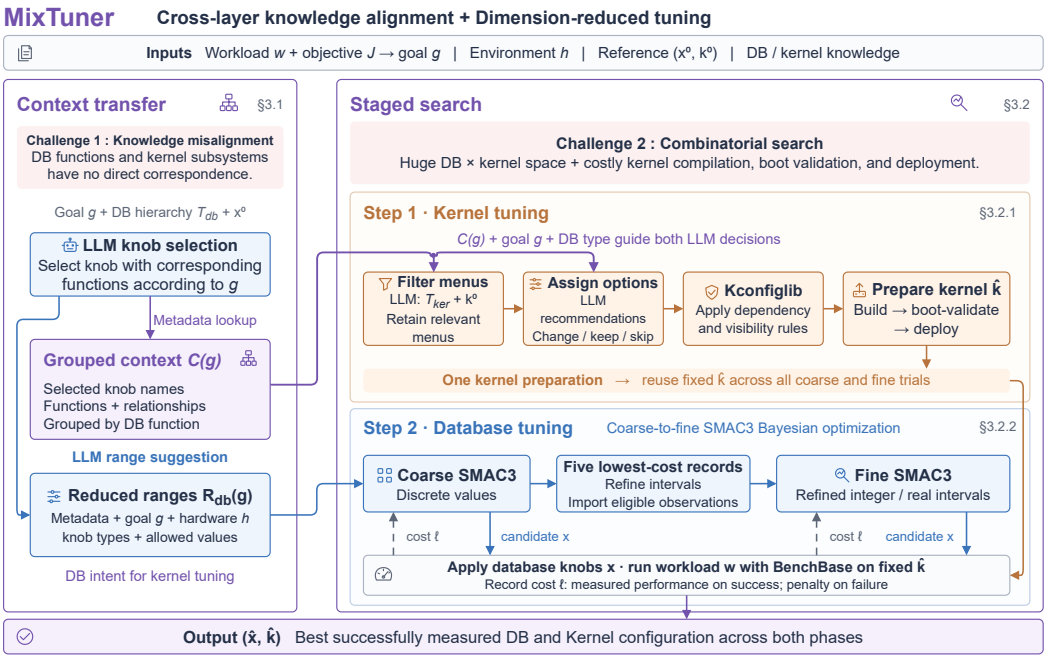


Figure 2: Overview of MixTuner. Context transfer selects database knobs and produces grouped context. Staged search first configures and deploys a kernel using this context, then optimizes database configurations through coarse-to-fine Bayesian optimization on that fixed kernel.

3 Context Transfer and Staged Search

MixTuner consists of two components aligned with the challenges in Section 1. *Context transfer* addresses the knowledge-misalignment challenge by representing selected database knobs and their functions as context for interpreting kernel configuration knowledge. *Staged search* addresses the combinatorial-search challenge by narrowing the candidate space and separating tuning into two steps: kernel tuning uses this context to prepare a workload-relevant kernel, and database tuning optimizes knob values through repeated measurements on that fixed kernel. This schedule shares kernel preparation across database trials.

The inputs of MixTuner are workload w, hardware and fixed runtime environment h, performance objective J, and reference configurations $(\mathbf{x}^0, \mathbf{k}^0)$, as defined in Section 2. Database hierarchy and knowledge $\mathcal{T}_{\text{db}}$ and kernel hierarchy and knowledge $\mathcal{T}_{\text{ker}}$ provide configuration descriptions and their organization. We express the workload and objective as a natural-language goal g, such as increasing transaction throughput for the target workload.

Figure 2 shows the two components and their data flow. Context transfer selects knobs under goal g and produces grouped context $C(g)$ for kernel tuning, together with reduced ranges $\mathcal{R}_{\text{db}}(g)$ for database tuning. Staged search first combines $C(g)$ with Kconfig knowledge to filter menus and recommend options, then builds, boot-validates, and deploys the resulting kernel $\widehat{\mathbf{k}}$. Its second step searches database values within $\mathcal{R}_{\text{db}}(g)$ using coarse-to-fine Bayesian optimization. Measurement feedback updates this numerical search while $C(g)$ and $\widehat{\mathbf{k}}$ remain fixed. The output is the best successfully measured database configuration $\widehat{\mathbf{x}}$ paired with $\widehat{\mathbf{k}}$. Algorithm 1 in Appendix A.1 gives the complete procedure.

3.1 Context Transfer

Context transfer takes the goal g, database knowledge $\mathcal{T}_{\text{db}}$, reference values $\mathbf{x}^0$, and hardware information from h. It selects workload-relevant knobs and expresses their database functions in a form that can be supplied alongside kernel knowledge. The grouped context $C(g)$ guides kernel

tuning in Section 3.2.1, while the accompanying search ranges $\mathcal{R}_{\mathrm{db}}(g)$ restrict database tuning in Section 3.2.2 to the selected knobs and values.

The database hierarchy organizes configuration entries as a root, functional groups, and individual knobs. Each entry records its name, description, type, default, legal values, and reload or restart requirements.[1] An LLM groups the entries by function during knowledge preparation, and the resulting hierarchy can be reused across tuning runs. Appendix A.2 describes collection.

Goal-conditioned knob selection. Selection proceeds one functional group at a time. Each request supplies g, the group name, and its knob names with current or default values. The prompt asks the LLM to select adjustable parameters whose changes are likely to improve the stated objective and to exclude read-only metadata such as version numbers. It returns a list of selected names, or an empty list when no knob in the group is judged relevant. For example, the MySQL TPC-H prompt selects `innodb_buffer_pool_size` and `innodb_io_capacity` from an InnoDB group while omitting `innodb_version`. The judgment uses the goal, group context, parameter names, values, and the model's prior knowledge. It is a semantic relevance judgment made before workload measurements, rather than a measured importance score. Appendix C.2 gives the exact input and output format.

Two outputs from the selected names. The returned names identify entries in $\mathcal{T}_{\mathrm{db}}$. A metadata lookup attaches descriptions and documented relationships for kernel conditioning; a separate range request uses the selected entries to prepare numerical search. A group contributes only if at least one of its knobs is selected. Let x_i^c denote a selected knob from group c corresponding to coordinate i of $\mathbf{x}$, and let $S(g)$ collect the selected group–knob indices (c, i). We write $\mathrm{value}(x_i^c) = x_i$ for its assigned value.

The context supplied to kernel tuning contains information about the knobs and functional groups selected under goal g:

$$C(g) = \left\{ \left(c, \mathrm{info}(x_i^c)\right) \;\middle|\; (c,i) \in S(g) \right\}. \tag{3}$$

The set contains one entry per knob selected for g. Here, $\mathrm{info}(x_i^c)$ denotes the knob's configuration information, including its name, functional description, and documented relationships with other settings. Entries sharing c are supplied together as a functional group. For example, selecting PostgreSQL's `shared_buffers` and `work_mem` places both under `Memory`, together with information about their respective roles in shared buffering and query working memory. During kernel tuning, the LLM relates this grouped information to the menu descriptions in $\mathcal{T}_{\mathrm{ker}}$ to identify relevant mechanisms and recommend options that support the database functions being tuned.

The same selected knobs define $\mathcal{R}_{\mathrm{db}}(g)$ for the database-tuning step of staged search. The LLM uses each knob's documented metadata, goal g, and hardware information to suggest numerical bounds within the legal limits. The allowed values are expressed according to the knob type:

$$\mathrm{value}(x_i^c) \in R_i^c = \begin{cases} [l_i^c, u_i^c], & \text{continuous,} \\ [l_i^c, u_i^c] \cap \mathbb{Z}, & \text{integer,} \\ \{v_{i,1}^c, \ldots, v_{i,m_i}^c\}, & \text{Boolean or enumeration.} \end{cases} \tag{4}$$

For a Boolean or enumeration knob, $v_{i,1}^c, \ldots, v_{i,m_i}^c$ are its m_i allowed choices represented by the backend; a Boolean knob has two choices. The search ranges are recorded together with the selected knobs' groups, information, and types:

$$\mathcal{R}_{\mathrm{db}}(g) = \left\{ \left(c, \mathrm{info}(x_i^c), \mathrm{type}(x_i^c), R_i^c\right) \;\middle|\; (c,i) \in S(g) \right\}. \tag{5}$$

Each record describes one knob selected for g. In the numerical search below, $\mathcal{R}_{\mathrm{db}}(g)$ denotes the configuration space induced by these records: selected knobs take values from their ranges, while unselected knobs retain their reference values. These ranges provide the starting point for measurement-driven database tuning, which determines the values of the selected knobs x_i^c. The LLM determines which functional groups c contribute selected knobs and generates $C(g)$ and $\mathcal{R}_{\mathrm{db}}(g)$ through its selection and range judgments, using the documented information in $\mathcal{T}_{\mathrm{db}}$. Appendices C.2 and C.3 give the corresponding prompts.

[1] MySQL entries are collected from its reference manual; PostgreSQL entries combine server metadata from `pg_settings` with documentation (The PostgreSQL Global Development Group, 2026b).

3.2 STAGED SEARCH

Staged search uses $C(g)$ and $\mathcal{R}_{\mathrm{db}}(g)$ to turn the joint configuration problem into two ordered tuning steps. *Kernel tuning* narrows Kconfig to relevant menus, recommends option values, and deploys the resulting kernel. *Database tuning* then searches the reduced knob space on that kernel. Preparing the kernel once avoids repeating compilation and deployment for each database trial, while coarse-to-fine search concentrates measurements within the selected database ranges.

3.2.1 STEP 1: KERNEL TUNING

Kernel tuning takes $C(g)$, goal g, the database type, kernel knowledge $\mathcal{T}_{\mathrm{ker}}$, and reference configuration $\mathbf{k}^0$. It uses the grouped context as a shared input to two decisions: which menu branches to inspect, and which visible options to change. For the memory-knob example above, the descriptions of shared buffering and query working memory connect the selected database knobs to the functions described in kernel memory-management menus. The LLM makes this relevance judgment from the two sets of descriptions; Kconfig subsequently determines which recommended assignments are permitted.

The kernel hierarchy and knowledge $\mathcal{T}_{\mathrm{ker}}$ are parsed from Linux Kconfig using Kconfiglib (Kconfiglib Contributors, 2026). They comprise subsystem menus, choices, and configuration symbols with help text and metadata. Starting with the top-level menus, the LLM receives the menus in the current scope together with g, the database type, and $C(g)$. It retains menus that may support the workload and prunes those judged irrelevant. Traversal continues into retained branches up to a configured depth. This filtering produces $\mathcal{T}'_{\mathrm{ker}} \subseteq \mathcal{T}_{\mathrm{ker}}$, the subset of kernel knowledge retained for option recommendations.

Within the retained branches of $\mathcal{T}'_{\mathrm{ker}}$, the LLM examines visible options in batches. Each request supplies option names, types, current values, help text, and assignable values, along with the same g and $C(g)$ used for menu filtering. The model recommends changes, keeps current values, or skips options. Database context therefore influences both where the tuner looks and what it recommends within those regions. Kconfiglib applies the recommendations under Kconfig's dependency and visibility rules (The Linux Kernel Contributors, 2026), starting from $\mathbf{k}^0$, and exports the resulting configuration $\widehat{\mathbf{k}}$. The exported configuration is passed to compilation, boot validation, and deployment outside the configuration-generation routine. Once the kernel configured by $\widehat{\mathbf{k}}$ is running and the workload can execute, database tuning begins. Appendix A.5 gives implementation details, and Appendices C.4 and C.5 give the two prompts.

3.2.2 STEP 2: DATABASE TUNING

Database tuning combines SMAC3 Bayesian optimization (Lindauer et al., 2022) with coarse-to-fine search (Lao et al., 2024) to optimize knob values on the kernel $\widehat{\mathbf{k}}$ deployed in the first step. Starting from $\mathcal{R}_{\mathrm{db}}(g)$, coarse search explores representative value combinations, and fine search refines numerical regions identified by the coarse measurements.

Coarse exploration. Coarse search restricts each numerical knob to a finite set of representative values spanning its suggested range. SMAC3 searches combinations of these values across the selected knobs in $\mathcal{R}_c$. For example, an integer knob $x_i \in [0, 80] \cap \mathbb{Z}$ can use $\{0, 20, 30, 40, 50, 60, 80\}$ as its coarse candidates: each evaluated configuration assigns x_i one of these seven values. This phase identifies promising combinations at a limited numerical resolution. Appendix A.3 gives the candidate construction and interval refinement rules.

Fine refinement. Fine search replaces each numerical candidate set with an interval derived from the five lowest-cost coarse configurations, forming $\mathcal{R}_f$. SMAC3 can select any integer in that interval for integer knobs, or a continuous value for real-valued knobs. Continuing the example, if all five retained configurations assign $x_i = 40$ while differing in other knobs, the refinement rule gives $x_i \in [32, 48] \cap \mathbb{Z}$. Fine search can therefore test values such as 37 and 43, which are absent from the coarse candidates. Boolean and enumeration choices remain unchanged across both phases.

Execution and feedback. Execution begins with coarse search over $\mathcal{R}_c$. SMAC3 proposes a configuration $\mathbf{x}_{t+1}$ using an initial design or the accumulated coarse history H_c. The database adapter applies the configuration, and BenchBase (Carnegie Mellon Database Group, 2026) executes workload w on the fixed kernel $\widehat{\mathbf{k}}$ in environment h. In either search phase, the execution result is converted into the minimization cost

$$\ell_{t+1} = \begin{cases} -\widehat{J}(\mathbf{x}_{t+1}, \widehat{\mathbf{k}}; w, h), & \text{if execution succeeds,} \\ P, & \text{if execution fails.} \end{cases} \tag{6}$$

Here, $P = 10^{10}$ penalizes failures in configuration application or workload execution. SMAC3 adds the configuration–cost pair to H_c and uses the updated history to guide further proposals within $\mathcal{R}_c$. This proposal–execution–feedback loop continues until the coarse trial limit B_c is reached.

The search then switches to fine optimization. The five lowest-cost coarse records determine the refined domain $\mathcal{R}_f$ as described above. Eligible records are imported into the fine history H_f with their measured costs, so fine search starts with feedback from coarse search. A new SMAC3 optimizer searches $\mathcal{R}_f$. Each proposed configuration is applied and evaluated using the same procedure, and its cost is added to H_f to guide subsequent fine search proposals. Thus, both the reused coarse measurements and new fine measurements inform the search. Fine optimization stops when H_f reaches the fine trial limit B_f, including the imported records.

Finally, we return the configuration $\widehat{\mathbf{x}}$ with the best successful measurement across both phases, together with $\widehat{\mathbf{k}}$. If neither phase produces a successful trial, the procedure reports search failure. Appendix A.4 describes the SMAC3 optimization procedure.

4 Experiments

4.1 Experimental Setup

Benchmarks and scenarios. We use BenchBase (Difallah et al., 2013; Carnegie Mellon Database Group, 2026) to run TPC-C and TPC-H on MySQL 8.4.11 and PostgreSQL 16.14. TPC-C represents transaction processing, whereas TPC-H supplies analytical queries. The main evaluation covers the Cartesian product of two databases, two workloads, scale factors $\mathrm{SF} \in \{0.1, 1, 10\}$, and configured terminal counts $T \in \{1, 6, 12\}$. SF is the workload generator's data-scale parameter; T is the number of configured client terminals. More settings are given in Appendix B.1.

Baselines. *Reference Config* denotes the officially recommended database–kernel pair $(\mathbf{x}^0, \mathbf{k}^0)$ defined in Section 2. It retains the vendor-provided database settings and the operating-system distribution's kernel configuration in use before tuning, with no LLM-based or measurement-driven optimization. We organize tuning baselines by the layer they tune: database-only methods include GP-Tuner (Lao et al., 2024), Lasso-Rank Pruning (Van Aken et al., 2017; 2021), and SysInsight (Zhang et al., 2026); Kernel-only methods include AutoOS (Chen et al., 2024), BYOS (Lin et al., 2026b), and TuneAgent (Lin et al., 2026a). MixTuner uses GPT-5.5 as its default base LLM. We also evaluate DeepSeek-V4-pro and Qwen-3.8-max as alternative base LLMs for MixTuner. More details are given in Appendices B.7 and B.6.

Platform and implementation. Experiments run on an Intel Xeon Gold 6348 server at 2.60 GHz, with 28 physical cores and 56 logical processors per CPU. The server has 512 GB of RAM and 512 GB of disk storage and runs Ubuntu 24.04.4 LTS. Kernel menu filtering uses a depth of two, and option assignment processes batches of ten. Database tuning uses coarse-to-fine SMAC3 optimization (Lindauer et al., 2022), with fine search initialized from the five best coarse observations. More details are given in Appendices A and B.1.

Metrics. We evaluate TPC-C by throughput q (requests/s) and TPC-H by per-run P95 request latency l (seconds), using BenchBase metrics (Difallah et al., 2013; Carnegie Mellon Database Group, 2026). For N measured requests over Δt seconds and latencies $\{\tau_i\}_{i=1}^{N}$ converted to seconds,

$$q = \frac{N}{\Delta t}, \qquad l = Q_{0.95}\big(\{\tau_i\}_{i=1}^{N}\big), \tag{7}$$

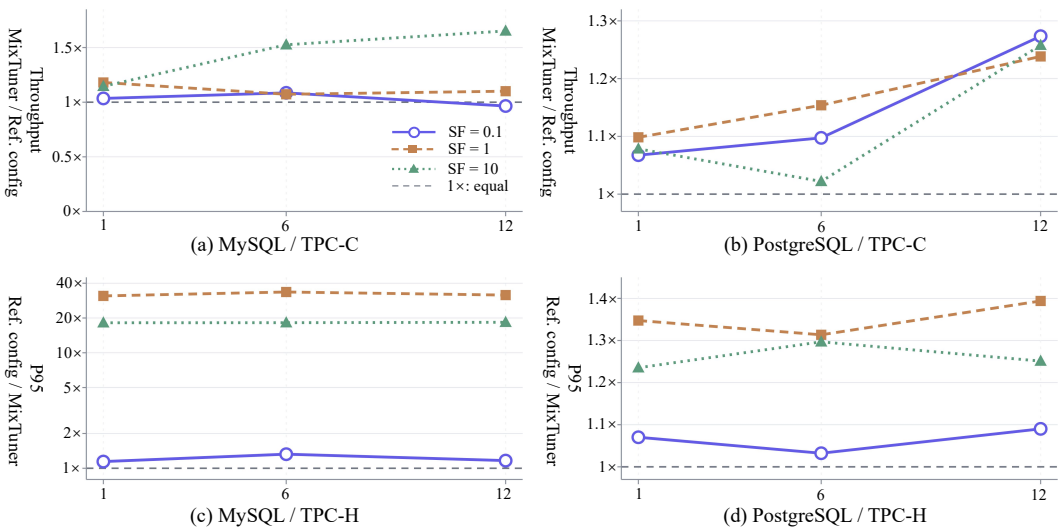


Figure 3: Performance ratios for (a) MySQL/TPC-C, (b) PostgreSQL/TPC-C, (c) MySQL/TPC-H, and (d) PostgreSQL/TPC-H. The vertical axes show q_{MixTuner}/q_0 for TPC-C and l_0/l_{MixTuner} for TPC-H, with Reference Config at $1\times$.

where $Q_{0.95}$ denotes the empirical 95th percentile. Higher throughput and lower P95 latency indicate better performance.

4.2 Main Results

MixTuner outperforms Reference Config. Figure 3 reports performance ratios with Reference Config normalized to $1\times$. On TPC-H, MixTuner reduces MySQL P95 latency by up to $33.58\times$ (163.55 s to 4.87 s at SF=1 and $T = 6$) and PostgreSQL P95 latency by up to $1.39\times$. On TPC-C, throughput reaches up to $1.65\times$ Reference Config on MySQL and $1.27\times$ on PostgreSQL. The only regression is a 3.5% throughput decrease for MySQL TPC-C at SF=0.1 and $T = 12$; Appendix B.2 provides all measurements. Appendix B.3 gives the MixTuner search trajectories for all four tasks.

Table 1: Method comparison at SF=1 and $T = 6$: mean ($\pm$ standard deviation) over three repetitions, rounded to two decimals. D and K denote database and kernel tuning; D+K denotes both.

		TPC-C throughput (req/s) ↑		TPC-H P95 (s) ↓	
Method	Scope	MySQL	PostgreSQL	MySQL	PostgreSQL
Reference Config	None	63.67 ($\pm$1.59)	167.02 ($\pm$4.18)	163.55 ($\pm$4.91)	2.36 ($\pm$0.19)
GPTuner	D	66.82 ($\pm$2.10)	187.22 ($\pm$3.49)	5.00 ($\pm$0.60)	1.93 ($\pm$0.14)
Lasso-Rank Pruning	D	65.11 ($\pm$1.69)	183.29 ($\pm$4.82)	5.03 ($\pm$0.67)	2.31 ($\pm$0.10)
SysInsight	D	65.02 ($\pm$1.63)	156.90 ($\pm$4.51)	5.08 ($\pm$0.61)	1.90 ($\pm$0.12)
AutoOS	K	63.98 ($\pm$2.73)	168.89 ($\pm$2.93)	148.32 ($\pm$2.97)	2.29 ($\pm$0.18)
BYOS	K	59.87 ($\pm$1.68)	160.51 ($\pm$4.33)	168.71 ($\pm$3.57)	2.35 ($\pm$0.20)
TuneAgent	K	63.24 ($\pm$1.64)	169.19 ($\pm$3.23)	159.23 ($\pm$2.94)	2.12 ($\pm$0.16)
Vanilla LLM Tuning	D+K	64.21 ($\pm$2.01)	169.89 ($\pm$4.25)	145.91 ($\pm$2.92)	2.19 ($\pm$0.15)
AutoOS + GPTuner	D+K	65.12 ($\pm$1.50)	186.75 ($\pm$2.30)	5.08 ($\pm$0.56)	2.02 ($\pm$0.13)
AutoOS + SysInsight	D+K	66.10 ($\pm$1.97)	188.86 ($\pm$2.34)	5.21 ($\pm$0.57)	2.12 ($\pm$0.13)
MixTuner(ours)	D+K	**68.28** ($\pm$**1.47**)	**192.72** ($\pm$**3.85**)	**4.87** ($\pm$**0.49**)	**1.80** ($\pm$**0.09**)

MixTuner outperforms existing tuning methods. Table 1 shows that MixTuner has the best mean performance on all four tasks. Relative to the strongest baseline in each column, it improves mean TPC-C throughput by 2.18% on MySQL and 2.04% on PostgreSQL, and reduces mean TPC-H P95 by 2.60% and 5.26%, respectively. Database-only tuning reduces MySQL TPC-H P95 from 163.55 s to 5.00–5.08 s, while MixTuner further reduces it to 4.87 s. AutoOS + GPTuner and AutoOS + SysInsight do not combine database knob knowledge with kernel-domain expertise. MixTuner outperforms both pipelines on all four tasks, with 2.04–4.85% higher throughput and 4.13–15.09% lower P95 latency. This gap supports conditioning kernel choices on database-specific

functional requirements. Appendix B.8 summarizes recorded tuning time and LLM API expenditure on MySQL/TPC-H and relates these costs to the performance results in Table 1.

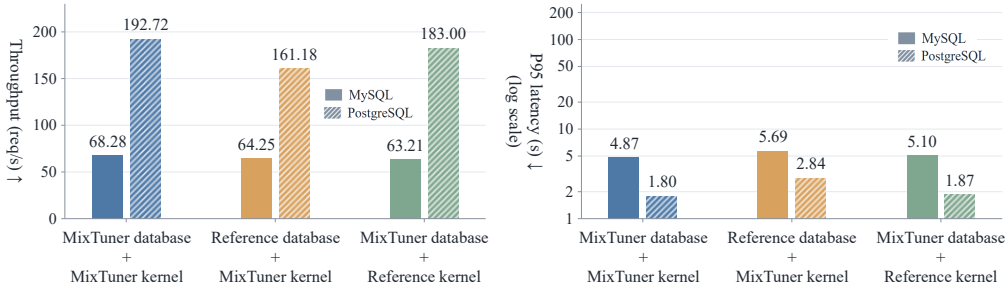
4.3 Ablation Studies

MixTuner achieves strong performance across the evaluated base models. We evaluate GPT-5.5, DeepSeek-V4-pro, and Qwen-3.8-max in the complete MixTuner workflow on MySQL TPC-H at SF=1 and $T = 6$. Table 2 reports the search-best P95 latency and its increase over GPT-5.5. DeepSeek-V4-pro and Qwen-3.8-max yield latencies only 1.19% and 1.91% higher than GPT-5.5, respectively, indicating similar performance across all three models. More details are given in Appendix B.6. We also ablate the database context supplied to the kernel tuner to evaluate context transfer and functional grouping in Appendix B.5.

Table 2: MixTuner's base-model comparison on MySQL TPC-H. Relative changes use unrounded measurements.

Base model	P95 (s) ↓	ΔP95 (%)
GPT-5.5	**4.87**	0.00
DeepSeek-V4-pro	4.93	+1.19
Qwen-3.8-max	4.96	+1.91

Tuning both database and kernel settings yields the best performance. Figure 4 compares full MixTuner with variants retaining either the reference kernel or the reference database settings. Against the reference-kernel variant, full MixTuner increases TPC-C throughput by 8.0% on MySQL and 5.3% on PostgreSQL, while reducing TPC-H P95 by 3.8–4.5%. Against the reference-database variant, it increases TPC-C throughput by 6.3–19.6% and reduces TPC-H P95 from 5.69 s to 4.87 s on MySQL (14.41%) and from 2.84 s to 1.80 s on PostgreSQL.



(a) TPC-C (b) TPC-H

Figure 4: Database–kernel configuration variants at SF=1 and $T = 6$: (a) TPC-C throughput and (b) TPC-H P95 latency.

5 Related Work

Kernel tuning. Wayfinder uses neural search over compile-time, boot-time, and runtime parameters (Jung et al., 2026), while KconfigTune combines dependency repair with random-forest Bayesian optimization (Sun et al., 2026). Among LLM-guided methods, AutoOS explores the dynamic Kconfig tree (Chen et al., 2024), BYOS connects operating-system concepts and options through a dual-layer knowledge graph (Lin et al., 2026b), and TuneAgent learns configuration policies through correctness-oriented warm-up and performance-oriented exploration (Lin et al., 2026a).

Database tuning. OtterTune (Van Aken et al., 2017; 2021) ranks knobs and transfers observations, LlamaTune (Kanellis et al., 2022) reduces search dimensionality, and ESTune (Yue et al., 2026) stops unpromising evaluations early. DB-BERT (Trummer, 2022) extracts tuning hints from text, while SysInsight (Zhang et al., 2026) derives tuning rules from DBMS source code and runtime observations. GPTuner (Lao et al., 2024) and MCTuner (Yan et al., 2025) guide search with LLMs, whereas E2ETune (Huang et al., 2025) generates configurations. LLAMBO (Liu et al., 2024) applies LLMs to general black-box optimization.

6 Conclusion

MixTuner addresses database–kernel co-tuning by carrying database knowledge into kernel configuration, then using workload measurements to select database values. Experiments show that MixTuner outperforms existing tuning methods.

AI USE STATEMENT

We used generative AI to assist with drafting the initial manuscript and searching for related literature. Language models are also an integral component of MixTuner, as described in Section 3. All AI-assisted work was reviewed and further refined by the authors. The authors are responsible for the final manuscript and its supporting evidence.

ETHICS STATEMENT

This work studies configuration search for database benchmark workloads. The evaluation protocol treats transaction isolation and durability guarantees as part of the workload definition, because relaxing these guarantees changes the service being measured. Kernel recommendations require testing in the target execution environment before their performance can be assessed. Model inputs and released artifacts require review for credentials, identifying host information, and sensitive workload content.

REPRODUCIBILITY STATEMENT

Algorithm 1 in Appendix A.1 specifies the tuning workflow. Appendix A describes knowledge collection, numerical search, and kernel configuration, and Appendix C gives the prompt templates. Section 4.1 defines the comparison protocol and aggregation. Tables 3 and 4 describe the run specifications and result-source fields; Appendix B.2 gives all 36 paired measurements. The method's source code is provided in the supplementary material to support reproducibility.

REFERENCES

Carnegie Mellon Database Group. BenchBase: Multi-DBMS SQL benchmarking framework via JDBC, 2026. URL https://github.com/cmu-db/benchbase. Accessed September 8, 2026.

Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, Qi Guo, and Yunji Chen. AutoOS: Make your OS more powerful by exploiting large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 7511–7525, 2024. URL https://proceedings.mlr.press/v235/chen24at.html.

Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. E2ETune: End-to-end knob tuning via fine-tuned generative language model. *Proceedings of the VLDB Endowment*, 18(13):5540–5554, 2025. doi: 10.14778/3773731.3773732. URL https://www.vldb.org/pvldb/vol18/p5540-huang.pdf.

Alexander Jung, Cezar Crăciunoiu, Nikolaos Karaolidis, Hugo Lefeuvre, Daniel Oñoro Rubio, Felipe Huici, Charalampos Rotsos, and Pierre Olivier. Wayfinder: Automated operating system specialization. In *Proceedings of the 21st European Conference on Computer Systems*, pp. 710–727. ACM, 2026. doi: 10.1145/3767295.3803589. URL https://doi.org/10.1145/3767295.3803589.

Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. LlamaTune: Sample-efficient DBMS configuration tuning. *Proceedings of the VLDB Endowment*, 15(11):2953–2965, 2022. doi: 10.14778/3551793.3551844.

Kconfiglib Contributors. Kconfiglib: A Kconfig implementation in Python, 2026. URL https://github.com/zephyrproject-rtos/Kconfiglib. Zephyr Project repository; accessed September 15, 2026.

Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. GPTuner: A manual-reading database tuning system via GPT-guided bayesian optimization. *Proceedings of the VLDB Endowment*, 17(8):1939–1952, 2024. doi: 10.14778/3659437.3659449. URL https://www.vldb.org/pvldb/vol17/p1939-tang.pdf.

Hongyu Lin, Yuchen Li, Haoran Luo, Zhenghong Lin, Libo Zhang, Mingjie Xing, and Yanjun Wu. TuneAgent: Agentic operating system kernel tuning with reinforcement learning. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 3062–3073. ACM, 2026a. doi: 10.1145/3770855.3817987. URL https://doi.org/10.1145/3770855.3817987.

Hongyu Lin, Yuchen Li, Haoran Luo, Kaichun Yao, Libo Zhang, Zhenghong Lin, Mingjie Xing, Yanjun Wu, and Carl Yang. BYOS: Knowledge-driven large language models bring your own operating system more excellent. *arXiv preprint arXiv:2503.09663*, 2026b. URL https://arxiv.org/abs/2503.09663v3. Version 3, revised February 12, 2026; first submitted in 2025.

Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. URL https://www.jmlr.org/papers/v23/21-0888.html.

Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large language models to enhance bayesian optimization. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2402.03921.

Oracle. MySQL 8.4 reference manual: Using asynchronous I/O on linux, 2026. URL https://dev.mysql.com/doc/refman/8.4/en/innodb-linux-native-aio.html. Accessed September 15, 2026.

Ying Sun, Fangqi Bi, Jiatai He, Sheng Qu, Yanjun Wu, and Pengpeng Hou. KconfigTune: Automatic performance tuning for Linux kernel configuration. *IEEE Transactions on Computers*, 75(3): 1000–1013, 2026. doi: 10.1109/TC.2025.3644413. URL https://doi.org/10.1109/TC.2025.3644413.

The Linux Kernel Contributors. Linux v6.8: Initialization configuration (init/Kconfig), 2024. URL https://github.com/torvalds/linux/blob/v6.8/init/Kconfig. AIO configuration symbol; accessed September 15, 2026.

The Linux Kernel Contributors. Kconfig language. Linux kernel documentation, 2026. URL https://docs.kernel.org/kbuild/kconfig-language.html. Accessed September 8, 2026.

The PostgreSQL Global Development Group. PostgreSQL 16 documentation: Managing kernel resources, 2026a. URL https://www.postgresql.org/docs/16/kernel-resources.html. Accessed September 8, 2026.

The PostgreSQL Global Development Group. PostgreSQL 16 documentation: pg_settings, 2026b. URL https://www.postgresql.org/docs/16/view-pg-settings.html. Accessed September 15, 2026.

Immanuel Trummer. DB-BERT: A database tuning tool that "reads the manual". In *Proceedings of the 2022 international conference on management of data*, pp. 190–203, 2022. doi: 10.1145/3514221.3517843.

Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pp. 1009–1024, 2017. doi: 10.1145/3035918.3064029.

Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment*, 14 (7):1241–1253, 2021. doi: 10.14778/3450980.3450992. URL https://www.vldb.org/pvldb/vol14/p1241-aken.pdf.

Zihan Yan, Rui Xi, and Mengshu Hou. MCTuner: Spatial decomposition-enhanced database tuning via LLM-guided exploration. *Proceedings of the ACM on Management of Data*, 3(6):342:1–342:25, 2025. doi: 10.1145/3769807. URL https://doi.org/10.1145/3769807.

Zhongwei Yue, Jun-Peng Zhu, Peng Cai, Xuan Zhou, Quanqing Xu, and Chuanhui Yang. ESTune: Bayesian uncertainty-guided early stopping for database configuration tuning. *Proceedings of the ACM on Management of Data*, 4(1):35:1–35:26, 2026. doi: 10.1145/3786649. URL https://doi.org/10.1145/3786649.

Xinyi Zhang, Tiantian Chen, Zhentao Han, Zhaoyan Hong, Wei Lu, Sheng Wang, Mo Sha, Anni Wang, Shuang Liu, Yakun Zhang, Feifei Li, and Xiaoyong Du. Why database manuals are not enough: Efficient and reliable configuration tuning for DBMSs via code-driven LLM agents. *Proceedings of the VLDB Endowment*, 19(6):1358–1371, 2026. doi: 10.14778/3797919.3797940. URL https://www.vldb.org/pvldb/vol19/p1358-zhang.pdf.

A Implementation Details

A.1 Complete Tuning Algorithm

Algorithm 1 gives the complete workflow described in Section 3. Context transfer prepares the grouped database context and search ranges. Staged search then performs kernel tuning followed by database tuning, reusing the deployed kernel throughout coarse and fine optimization.

Algorithm 1 MixTuner: context transfer and staged search

Require: Workload w, environment h, objective J, knowledge $\mathcal{T}_{\mathrm{db}}, \mathcal{T}_{\mathrm{ker}}$, reference settings $(\mathbf{x}^0, \mathbf{k}^0)$, budgets B_c, B_f

Ensure: Best successfully measured pair $(\widehat{\mathbf{x}}, \widehat{\mathbf{k}})$, or search failure

Express (w, J) as the natural-language goal g
Context transfer: construct grouped context and database search ranges
for all functional groups c in $\mathcal{T}_{\mathrm{db}}$ **do**
  Use the LLM to select knobs x_i^c relevant to g using $(g, c, \mathbf{x}^0)$
end for
Collect selected group–knob indices in $S(g)$; omit groups with no selected knobs
Construct $C(g)$ from selected knob information and groups by Equation 3
Use the LLM to suggest numerical ranges R_i^c under $(\mathcal{T}_{\mathrm{db}}, g, h)$; retain legal categorical choices
Construct $\mathcal{R}_{\mathrm{db}}(g)$ from groups, knob information, types, and ranges by Equation 5
Staged search, step 1: kernel tuning conditioned on $C(g)$
$\mathcal{T}'_{\mathrm{ker}} \leftarrow \mathrm{PruneMenus}(\mathcal{T}_{\mathrm{ker}}, g, C(g), \mathbf{k}^0)$
$\widehat{\mathbf{k}} \leftarrow \mathrm{AssignKernelOptions}(\mathcal{T}'_{\mathrm{ker}}, g, C(g), \mathbf{k}^0)$
Build, boot-validate, and deploy $\widehat{\mathbf{k}}$; proceed once the workload can execute
Staged search, step 2: database tuning on the fixed kernel $\widehat{\mathbf{k}}$
$\mathcal{R}_c \leftarrow \mathrm{Coarsen}(\mathcal{R}_{\mathrm{db}}(g))$; $H_c, H_f \leftarrow \varnothing$
for $p \in (c, f)$, in order **do**
  if $p = f$ **then**
    $Q_c \leftarrow$ five records in H_c with the smallest execution costs ℓ
    $\mathcal{R}_f \leftarrow \mathrm{RefineNumericRanges}(Q_c, \mathcal{R}_{\mathrm{db}}(g))$
    $H_f \leftarrow \{(\mathbf{x}, \ell) \in Q_c : \mathbf{x} \in \mathcal{R}_f\}$
  end if
  Initialize a SMAC3 optimizer over $\mathcal{R}_p$ using history H_p
  while $|H_p| < B_p$ **do**
    $t \leftarrow |H_p|$; $H_t \leftarrow H_p$
    Use initial design or SMAC3 with H_t to propose $\mathbf{x}_{t+1}$ by Equation 10
    Execute $(\mathbf{x}_{t+1}, \widehat{\mathbf{k}})$ under (w, h); record the outcome and measured $\widehat{J}$ on success
    Compute ℓ_{t+1} by Equation 6; update H_p by Equation 11
  end while
end for
$\mathcal{X}_{\mathrm{succ}} \leftarrow$ successfully evaluated configurations from both phases; return failure if empty
Select $\widehat{\mathbf{x}}$ from $\mathcal{X}_{\mathrm{succ}}$ by Equation 12
return $(\widehat{\mathbf{x}}, \widehat{\mathbf{k}})$

A.2 Knowledge Base Collection

The MySQL collector reads the 8.4 reference manual and extracts descriptions and tabulated attributes, including type, default, numerical bounds, and dynamic mutability. When structured extraction fails, an LLM parses the description into the same schema using the template in Appendix C.1. The following entry illustrates the stored format, with a documented bound rather than a recommended tuning range:

```
{
  "name": "binlog_cache_size",
  "description": "Command-Line Format: --binlog-cache-size=# ...",
  "type": "NUMERIC",
```

```
  "default_value": "32768",
  "is_dynamic": true,
  "min_value": 4096,
  "max_value": 18446744073709547520,
  "valid_options_or_range": null
}
```

For PostgreSQL, the collector queries structured server metadata (The PostgreSQL Global Development Group, 2026b):

```
SELECT name, setting, unit, category, short_desc, extra_desc,
       context, vartype, source, min_val, max_val, enumvals,
       boot_val, reset_val, pending_restart
FROM pg_settings ORDER BY name;
```

Descriptions from the corresponding documentation supplement the short descriptions. Type and unit fields determine how numerical values are interpreted, and mutability fields determine how changes are applied. LLM grouping then assigns knob leaves to functional categories. Metadata inferred from prose requires comparison with the target server's actual parameter definitions.

Kernel knowledge combines parsed Kconfig entries with supplementary Linux Kernel Driver Database information. The implementation uses the Zephyr Project repository of Kconfiglib (Kconfiglib Contributors, 2026). Kernel prompts receive information about the selected database knobs grouped by function, including their names, functional descriptions, and documented relationships with other settings. Pruning records contain the menu scope and the model's relevance explanation, allowing a decision to be inspected alongside the input that produced it.

A.3 Numerical Search Details

For a numerical interval $[l_i, u_i]$, let V_i contain supplied representative values, or the internal quartiles when none are supplied. For each $v \in V_i$, the coarse candidate generator adds

$$v + \alpha(u_i - v) \quad \text{and} \quad v + \alpha(l_i - v), \qquad \alpha \in \{0, 0.25, 0.5\}.$$

It also includes the default and interval endpoints. Values are clipped to $[l_i, u_i]$ and deduplicated. Integer candidates use integer conversion; real-valued candidates are rounded to four decimal places. Boolean and enumeration parameters use categorical domains. The implementation preset configures 30 coarse trials, including a Latin-hypercube initial design of 10 configurations.

Fine search uses the five best coarse configurations. Let a_i and b_i be their minimum and maximum values for numerical knob i. The refinement rule is

$$\delta_i = \max\{0.5(b_i - a_i),\ 0.1(u_i - l_i)\}, \tag{8}$$

$$[l_i^f, u_i^f] = [\max(l_i, a_i - \delta_i),\ \min(u_i, b_i + \delta_i)]. \tag{9}$$

Integer bounds use integer conversion before comparison with the original endpoints. The refined interval is used when its lower bound is smaller than its upper bound; otherwise the previous interval is retained. If that interval is also degenerate, the current backend extends the upper bound by one for an integer or by 0.01 for a real-valued knob. Such a fallback changes the effective search bounds and must be recorded as part of the run specification. Fine search retains categorical choices and configures 50 trials, importing the five best coarse observations into its history. Imported observations are reused measurements.

The database adapter applies settings according to mutability or PostgreSQL context, requesting reloads or restarts where appropriate. The benchmark parser reads throughput in requests/s and P95 latency in microseconds. The minimization cost is negative throughput or P95 latency, and an application or execution failure receives the backend penalty 10^{10}. Performance summaries use successful measurements and preserve failure status separately. Effective settings must be captured in the benchmark's database sessions: setting a parameter on the tuner's own connection does not establish that it affected the workload connections.

A.4 SMAC3 BAYESIAN OPTIMIZATION

We use SMAC3 (Lindauer et al., 2022) to select database configurations in both search phases. Each phase creates a separate optimizer through `HyperparameterOptimizationFacade`, using its own domain $\mathcal{R} \in \{\mathcal{R}_c, \mathcal{R}_f\}$ and evaluation history. The workload, objective, and deployed kernel remain the same across the two phases.

Initialization. The coarse phase uses a Latin-hypercube initial design configured for 10 configurations. The fine phase uses the default configuration of its refined search space as its initial design and imports eligible records from the five lowest-cost coarse observations. Imported configurations retain their measured costs, providing observations for the new optimizer without repeating their workload executions.

Surrogate and candidate selection. Let $H_t = \{(\mathbf{x}_j, \ell_j)\}_{j=1}^{t}$ denote the current phase's history, including any imported records. SMAC3 models the observed costs with a random forest surrogate. For model-guided trials, it fits the surrogate to the encoded observations and approximately maximizes an expected-improvement acquisition function:

$$\begin{aligned} \mathcal{M}_t &= \mathrm{Fit}_{\mathrm{SMAC3}}(H_t), \\ \mathbf{x}_{t+1} &\approx \arg\max_{\mathbf{x} \in \mathcal{R}} a(\mathbf{x}; \mathcal{M}_t). \end{aligned} \tag{10}$$

Here, $\mathrm{Fit}_{\mathrm{SMAC3}}$ includes the facade's cost encoding and surrogate fitting, and a denotes its expected-improvement acquisition function. Candidate selection uses the predicted costs and uncertainty to assess potential improvement.

Feedback and termination. The proposed configuration is evaluated using the execution procedure in Section 3.2.2. Its cost, defined by Equation 6, is appended to the current history:

$$H_{t+1} = H_t \cup \{(\mathbf{x}_{t+1}, \ell_{t+1})\}. \tag{11}$$

Subsequent model-guided proposals use the updated history. Execution outcomes are recorded separately to identify successful configurations for the final output.

The phase limits are $B_c = 30$ and $B_f = 50$ history entries, including any imported observations. Failed trials consume the budget and contribute their penalty costs to the history. Let $\mathcal{X}_{\mathrm{succ}}$ contain the successfully evaluated configurations across both phases. The returned database configuration is

$$\widehat{\mathbf{x}} = \arg\max_{\mathbf{x} \in \mathcal{X}_{\mathrm{succ}}} \widehat{J}(\mathbf{x}, \widehat{\mathbf{k}}; w, h). \tag{12}$$

The search reports failure if $\mathcal{X}_{\mathrm{succ}}$ is empty.

A.5 KERNEL CONFIGURATION DETAILS

The default pruning depth is two menu levels. Each request contains the goal, database type, selected knob information grouped by functional category, and the available menus in the current scope. The prompt retains uncertain branches, and an unsuccessful pruning request retains its scope. Assignment then processes visible options in batches of ten, applies recommendations through Kconfiglib, and exports the resulting configuration. The exported effective values, including dependency-induced changes, identify the configuration passed to building and deployment. These execution steps precede the database optimization entry point.

Kernel support and runtime resource provisioning are separate conditions. PostgreSQL 16 on Linux normally uses a large anonymous `mmap` shared-memory region and POSIX semaphores, together with a small System V shared-memory allocation (The PostgreSQL Global Development Group, 2026a). Enabling `SYSVIPC` alone therefore does not determine a suitable `shared_buffers` range. Similarly, MySQL's native asynchronous I/O path requires the `libaio` library and is enabled by default on supported Linux configurations (Oracle, 2026). The introduction's disabled-AIO example illustrates how a kernel build option can constrain a database feature.

Table 3: Experimental coverage and run specifications. The method comparison covers all four tasks for every method in Table 1. The last column lists additional reproducibility records to document.

Study	Coverage	Additional reproducibility records
M1	MixTuner vs. Reference Config; 36 scenarios	Warm-up, state restoration and model settings
M2	MixTuner, nine tuning baselines and Reference Config; four tasks at SF=1, $T = 6$	Kernel artifacts, model settings and timings beyond Table 10
A1	Three models; MySQL TPC-H at SF=1, $T = 6$	Model snapshots, decoding, prompts, knowledge versions and optimizer seeds
A2	Full, Reference DB, Reference Kernel; four tasks at SF=1, $T = 6$	Reference-DB kernel identity, reference artifacts, fixed-configuration repetitions

B Experimental Configuration and Supplementary Results

B.1 Run Specifications and Result Sources

Workload execution. TPC-C runs for 60 s with NewOrder, Payment, OrderStatus, Delivery, and StockLevel weights of 45:43:4:4:4; its configured request-rate parameter is 10,000 and its batch size is 128. The weights specify the transaction mix, while the request-rate and batch-size values are inputs to the workload generator. Reported throughput is measured from completed requests and elapsed execution time. TPC-H uses Q1–Q22 with an unlimited rate and batch size 1,024. Its MySQL configuration specifies three serial phases covering all, even-numbered, and odd-numbered queries; PostgreSQL specifies one phase covering all queries.

The evaluation distinguishes the untuned Reference Config $(\mathbf{x}^0, \mathbf{k}^0)$, complete tuning methods, base-model replacements, and configuration-layer variants. Table 3 records their coverage and the remaining reproducibility fields. The current workloads use MySQL 8.4.11 and PostgreSQL 16.14; the logged generated kernels use Linux 6.8.12, and the reference-kernel ablation uses 6.8.0-136-generic.

The method comparison and configuration-layer ablation use SF=1 and $T = 6$. Table 1 reports all four database–workload pairs for Reference Config, MixTuner, and nine tuning baselines: GPTuner, Lasso-Rank Pruning, SysInsight, AutoOS, BYOS, TuneAgent, Vanilla LLM Tuning, AutoOS + GPTuner, and AutoOS + SysInsight. The configuration-layer ablation also covers all four tasks. The matched model ablation uses MySQL TPC-H at SF=1 and $T = 6$. Current runs use serializable transaction isolation.

The main MixTuner runs use GPT-5.5 for database knob selection, range suggestions, and kernel configuration. Kernel building and deployment precede SMAC3 database value search on the generated kernel.

Result selection and repeated measurements. For the 36-scenario evaluation in Appendix B.2, we report the best successful observation associated with the saved coarse and fine results for 32 scenarios. Four MySQL TPC-C scenarios instead use updated single BenchBase measurements under the same experimental settings; their records contain no per-evaluation search logs. Reference Config provides measurements of the untuned configuration for comparison in each scenario and performs no tuning search. The Reference DB variant also has one recorded configuration measurement per task. Appendix B.2 records the selection of searches from concatenated logs.

For Table 1, each method–task experiment, including Reference Config, is repeated three times at SF=1 and $T = 6$. Within each task, the repetitions use the same database version, workload definition, transaction isolation, and hardware environment. TPC-C retains the duration, transaction weights, request-rate parameter, and batch size specified above; TPC-H retains the query phases and workload-generator settings for the corresponding database. Each repetition yields one throughput measurement for TPC-C or one per-run P95 latency for TPC-H. These repeated measurements supply the statistics in Table 1; the single-run and search-best observations used elsewhere are identified separately in their respective appendices.

Table 4: Fields linking reported results to source measurements.

Field group	Contents
Identity	Experiment, method, database, workload, SF, terminals
Configuration	Kernel version, selected database configuration, effective parameter snapshot; model and prompt settings where available
Measurement	Source summary, metric, original and display units, success status
Search	Invocation boundaries, evaluation ID, new evaluations, earlier logged attempts
Aggregation	Search-best or single-run value; separately recorded fixed-configuration statistics when available

The current-run result index associates each measurement with its database, workload, SF, terminal count, source summary, result type, and selected search invocation where available. Table 4 defines the corresponding fields. For current TPC-H runs, the index retains the original microsecond value and the conversion to seconds.

B.2 All 36 Scenarios and Search Selection

Let $s = (\mathrm{DBMS}, \mathrm{workload}, \mathrm{SF}, T)$ identify an evaluation scenario, and let $q_0(s)$ and $l_0(s)$ denote its Reference Config throughput and P95 latency. Figure 3 and the tables below use the performance ratio

$$R(s) = \begin{cases} q_{\mathrm{MixTuner}}(s)/q_0(s), & \text{TPC-C}, \\ l_0(s)/l_{\mathrm{MixTuner}}(s), & \text{TPC-H}. \end{cases} \tag{13}$$

Each ratio pairs measurements at the same database system, workload, SF, and T, so $R(s) > 1$ indicates an improvement over the untuned pair $(\mathbf{x}^0, \mathbf{k}^0)$, and $R(s) < 1$ indicates a regression. Current TPC-H P95 values describe the recorded request-latency distribution and are converted from microseconds to seconds. Absolute performance values are displayed to two decimal places; the scenario ratios use unrounded measurements.

Tables 5 and 6 provide the absolute measurements underlying Figure 3. Reference Config reports measurements of the untuned database–kernel pair $(\mathbf{x}^0, \mathbf{k}^0)$ for each scenario. For the 32 scenarios reported from search logs, each MixTuner value is linked to the best successful observation of the invocation represented by its saved coarse and fine results. Earlier executions are accounted for separately below.

For MySQL TPC-C at SF$\in \{0.1, 1\}$ and $T \in \{6, 12\}$, we use updated single measurements recorded on September 23, 2026 under the same experimental settings. Their throughputs are 67.73 and 84.73 requests/s at SF=0.1 with $T = 6$ and $T = 12$, and 68.28 and 85.13 requests/s at SF=1 with $T = 6$ and $T = 12$, respectively. All four summaries record MySQL 8.4.11, serializable isolation, and successful completion; their workload configurations retain the 60 s duration, request-rate parameter 10,000, batch size 128, and transaction weights specified in Appendix B.1, which also describes the repetition protocol for Table 1. Each throughput equals the measured request count divided by the recorded elapsed time. These records replace the earlier searches in the current comparisons; the other 32 scenario values and all Reference Config measurements remain unchanged. The result index retains both generations of superseded searches separately. At SF=1 and $T = 6$, the Full configuration-layer variant uses this single measurement of 68.28 requests/s; the mean over three repetitions in Table 1 is also 68.28 requests/s, with a standard deviation of 1.47 requests/s. The updated records include benchmark summaries and database parameter snapshots, but no optimizer logs or kernel-build records, so they do not establish a new search trajectory or best-evaluation index.

Two historical source logs concatenate multiple optimizer invocations. The superseded MySQL TPC-C log at SF=0.1 and $T = 6$ contains an earlier invocation with no successful evaluations, followed by a second invocation; neither supplies the current table entry. For MySQL TPC-H at SF=1 and $T = 6$, a coarse-search invocation on September 17 is followed by the September 18 invocation represented by the saved coarse and fine results. The reported P95 is 4.87 s, also present in its individual benchmark summary. The current result index uses the supplied September 18 rerun directory, which contains the same selected invocation and value as its copy in the earlier export. The earlier summary export contains 5.13 s; the figures and tables consistently use the

Table 5: All TPC-C scenarios. Values are throughput in req/s. Reference Config reports measurements of the untuned pair $(\mathbf{x}^0, \mathbf{k}^0)$; MixTuner uses updated single measurements for MySQL at SF$\in \{0.1, 1\}$ and $T \in \{6, 12\}$, and search-best observations elsewhere. R follows Equation 13.

DBMS	SF	T	Reference Config	MixTuner	$R \uparrow$
MySQL	0.1	1	28.88	29.88	1.035
MySQL	0.1	6	62.40	67.73	1.085
MySQL	0.1	12	87.80	84.73	0.965
MySQL	1	1	24.77	29.23	1.180
MySQL	1	6	63.67	68.28	1.073
MySQL	1	12	77.35	85.13	1.101
MySQL	10	1	25.92	29.68	1.145
MySQL	10	6	121.68	185.63	1.526
MySQL	10	12	209.78	346.57	1.652
PostgreSQL	0.1	1	34.57	36.90	1.068
PostgreSQL	0.1	6	172.80	189.63	1.097
PostgreSQL	0.1	12	259.30	330.22	1.273
PostgreSQL	1	1	33.20	36.47	1.098
PostgreSQL	1	6	167.02	192.72	1.154
PostgreSQL	1	12	263.80	326.70	1.238
PostgreSQL	10	1	34.00	36.67	1.078
PostgreSQL	10	6	180.83	184.82	1.022
PostgreSQL	10	12	318.45	400.53	1.258

Table 6: All TPC-H scenarios. Values are P95 latency in s. Reference Config reports measurements of the untuned pair $(\mathbf{x}^0, \mathbf{k}^0)$; MixTuner is the best successful observation of the selected search. R follows Equation 13.

DBMS	SF	T	Reference Config	MixTuner	$R \uparrow$
MySQL	0.1	1	0.57	0.50	1.143
MySQL	0.1	6	0.68	0.51	1.324
MySQL	0.1	12	0.58	0.50	1.165
MySQL	1	1	159.68	5.14	31.059
MySQL	1	6	163.55	4.87	33.580
MySQL	1	12	161.67	5.12	31.583
MySQL	10	1	1033.66	56.84	18.185
MySQL	10	6	1032.87	56.71	18.213
MySQL	10	12	1030.95	56.16	18.359
PostgreSQL	0.1	1	0.18	0.17	1.070
PostgreSQL	0.1	6	0.17	0.16	1.032
PostgreSQL	0.1	12	0.18	0.16	1.090
PostgreSQL	1	1	2.41	1.79	1.347
PostgreSQL	1	6	2.36	1.80	1.314
PostgreSQL	1	12	2.46	1.76	1.394
PostgreSQL	10	1	21.72	17.58	1.235
PostgreSQL	10	6	22.03	16.99	1.297
PostgreSQL	10	12	21.86	17.48	1.251

individual summary associated with the selected invocation. Aggregate end-to-end search cost must also include the superseded searches and earlier executions.

B.3 Search Convergence at SF=10 and T=12

Fine search improves three of the four tasks, while MySQL TPC-C reaches its best value during coarse search. Figure 5 traces database value search at SF=10 and $T = 12$ on each task's fixed, generated kernel, using the same invocation as the corresponding main result. MySQL TPC-C reaches 346.57 requests/s at evaluation 11 and retains this incumbent through fine search. PostgreSQL TPC-C improves from a coarse-search incumbent of 348.78 to 400.53 requests/s, first reached at evaluation 57. MySQL TPC-H P95 decreases from 56.35 to 56.16 s, with the best value reached at evaluation 63; PostgreSQL TPC-H decreases from 17.60 to 17.48 s, first reached at evaluation 57.

The horizontal axis counts new evaluations within the selected invocation. All recorded evaluations succeed in each of these four runs. Each step retains the highest throughput or lowest P95 so far, including observations from both stages. The five coarse observations imported into fine search are not counted again. The final incumbents agree with the SF=10, $T = 12$ entries in Tables 5 and 6.

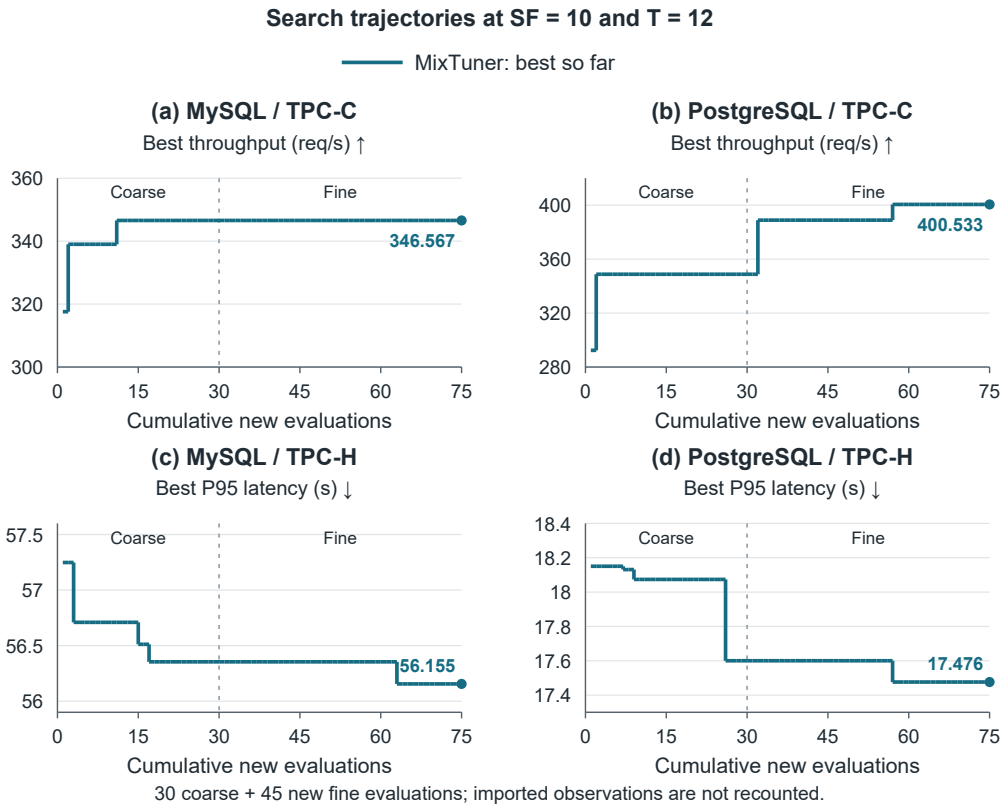


Figure 5: MixTuner search trajectories at SF=10 and $T = 12$. Solid steps show best-so-far performance versus cumulative new evaluations; the vertical dotted line at 30 marks the coarse-to-fine transition. TPC-C uses maximum throughput (requests/s, higher is better); TPC-H uses minimum P95 latency (s, lower is better). All panels use linear vertical axes. Each trajectory's final incumbent reproduces the corresponding main-experiment value.

B.4 Configuration-Layer Variants

Full MixTuner reuses the corresponding main-experiment result for the generated kernel K_* and selected database configuration D_*. For MySQL TPC-C, this is the updated single measurement of 68.28 requests/s; the other Full entries are search-best observations. Reference DB uses the rerun measurement of 5.69 s for MySQL TPC-H, giving Full a 14.41% P95 reduction to 4.87 s. Reference DB measures D_0 on a separately recorded MixTuner-generated kernel K_{A}. Reference Kernel holds the distribution kernel K_0 fixed and reruns database knob selection and value search, producing $D^*_{K_0}$. For MySQL TPC-C, Reference Kernel uses the rerun throughput of 63.21 requests/s, giving

Full an 8.0% throughput increase. Each variant retains one reference layer and tunes the other, whereas Reference Config retains both layers as $(D_0, K_0) = (\mathbf{x}^0, \mathbf{k}^0)$. These are workflow variants: the current Reference Kernel result is $(D^*_{K_0}, K_0)$, and Reference DB requires an additional artifact record to establish whether $K_{\mathrm{A}} = K_*$. Table 7 provides the absolute measurements.

Table 7: Configuration-layer measurements at SF=1 and $T = 6$. TPC-C values are throughput in requests/s; TPC-H values are P95 latency in seconds. Full uses an updated single measurement for MySQL TPC-C and search-best observations elsewhere. Reference Kernel uses a rerun measurement for MySQL TPC-C and search-best observations elsewhere; Reference DB is single-run.

DBMS	Workload	Full	Reference DB	Reference Kernel
MySQL	TPC-C ↑	68.28	64.25	63.21
PostgreSQL	TPC-C ↑	192.72	161.18	183.00
MySQL	TPC-H ↓	4.87	5.69	5.10
PostgreSQL	TPC-H ↓	1.80	2.84	1.87

The generated and reference kernels use different Linux builds. Their comparison therefore includes the full replacement of the kernel, together with database reoptimization. A frozen configuration interaction would instead require measurements of (D_0, K_0), (D_0, K_*), (D_*, K_0), and (D_*, K_*) using the same artifacts wherever a symbol recurs. Such an interaction is not estimated from the current workflow variants.

B.5 Database-Context Ablation

Table 8 reports the context ablation at SF=1 and $T = 6$. The three variants share the selected database knobs, search ranges, and optimizer settings. The intervention changes the context supplied to kernel menu filtering and option recommendation; database knob selection and range generation remain enabled in every variant.

Table 8: Database-context ablation at SF=1 and $T = 6$. The variants differ in the information supplied to kernel menu filtering and option recommendation.

		TPC-C throughput (req/s) ↑		TPC-H P95 (s) ↓	
Variant	Context supplied to the kernel tuner	MySQL	PostgreSQL	MySQL	PostgreSQL
No DB context	Optimization goal and database type	63.91	176.21	6.71	4.21
Flat DB context	Same knobs, descriptions, and relationships; no functional grouping	65.22	186.71	5.59	3.08
Grouped DB context (Full)	Complete grouped context $C(g)$	68.28	192.72	4.87	1.80

Results and interpretation. Grouped DB context achieves the best performance on all four tasks. Relative to No DB context, it increases TPC-C throughput by 6.84% on MySQL and 9.37% on PostgreSQL, and reduces TPC-H P95 latency by 27.42% and 57.24%, respectively. Flat DB context also improves all four metrics over No DB context, showing the benefit of supplying database knob information to the kernel tuner. Functional grouping provides a further gain while preserving the same knobs, descriptions, and relationships: Full improves throughput over Flat DB context by 4.69% on MySQL and 3.22% on PostgreSQL, and reduces P95 latency by 12.88% and 41.56%. The largest reduction occurs on PostgreSQL TPC-H, where P95 falls from 3.08 s with flat context to 1.80 s with grouped context. These results support both database-context transfer and its organization by functional group.

B.6 Base-Model Specifications and Run Selection

The model ablation compares GPT-5.5, DeepSeek-V4-pro, and Qwen-3.8-max within the complete MixTuner workflow. All three selected runs record MySQL 8.4.11, TPC-H, SF=1, $T = 6$, and serializable isolation. Each completes coarse and fine search, with five coarse observations imported into fine search. Table 9 identifies the generated kernels and the best observations. Each model contributes one selected search, so the reported differences describe observed optima rather than estimates of repeated-run variability.

Table 9: Artifacts and observations for the base-model results. All three rows use SF=1 and $T = 6$; their evaluation index counts new evaluations in the selected invocation, and each kernel is paired with its selected database configuration.

Base model	Kernel build	Best evaluation	P95 (s)
GPT-5.5	`6.8.12-mysql-tpch-v2`	54	4.87
DeepSeek-V4-pro	`6.8.12-mysql-tpch-ds`	65	4.93
Qwen-3.8-max	`6.8.12-mysql-tpch-qw`	1	4.96

The GPT-5.5 entry reuses the September 18 invocation selected in Appendix B.2. DeepSeek-V4-pro uses its September 6 invocation; Qwen-3.8-max uses its September 7–8 invocation. The best P95 values agree between each invocation's evaluation log, saved coarse/fine histories, and individual benchmark summary. Qwen-3.8-max reaches its recorded best at the first evaluation and retains it through fine search.

B.7 Baseline Comparison and Run Details

Table 1 compares nine tuning baselines with Reference Config and MixTuner on all four tasks at SF=1 and $T = 6$. Each method–task experiment is repeated three times under the experimental settings in Section 4.1. We report the mean and standard deviation of these three measurements, rounded to two decimal places. Relative gains for this method comparison are computed from the displayed means.

The database-only methods, GPTuner, Lasso-Rank Pruning, and SysInsight (Zhang et al., 2026), use the reference kernel $K_0 = \mathbf{k}^0$. The kernel-only methods, AutoOS, BYOS, and TuneAgent, retain the reference database settings $D_0 = \mathbf{x}^0$. Vanilla LLM Tuning directly uses an LLM to generate database and kernel configurations. AutoOS + GPTuner and AutoOS + SysInsight compose kernel tuning with database tuning on the resulting fixed kernel. Both pipelines tune the two layers (D+K) without transferring selected database-knob descriptions and functional relationships to the kernel tuner.

SysInsight baseline. SysInsight (Zhang et al., 2026) combines static analysis with LLM code reasoning to trace how database knobs affect execution. It converts the resulting hypotheses into quantitative tuning rules through association rule mining over tuning observations. During online tuning, runtime diagnosis identifies bottleneck functions and the associated knobs, whose values are adjusted using the retrieved rules. In the database-only baseline, this tuning operates on K_0; in AutoOS + SysInsight, it operates on the kernel produced by AutoOS.

Comparison across the four tasks. MixTuner achieves the highest mean TPC-C throughput and lowest mean TPC-H P95 latency in every column of Table 1. The strongest TPC-C baselines are GPTuner on MySQL (66.82 requests/s) and AutoOS + SysInsight on PostgreSQL (188.86 requests/s), compared with MixTuner's 68.28 and 192.72 requests/s. The strongest TPC-H baselines are GPTuner on MySQL (5.00 s) and SysInsight on PostgreSQL (1.90 s), compared with 4.87 s and 1.80 s for MixTuner.

Composing kernel and database tuners. On MySQL and PostgreSQL, respectively, AutoOS + GPTuner achieves TPC-C throughput of 65.12 and 186.75 requests/s and TPC-H P95 latency of 5.08 s and 2.02 s. AutoOS + SysInsight achieves 66.10 and 188.86 requests/s, with P95 latencies of 5.21 s and 2.12 s. Both pipelines remain below MixTuner on all four tasks. Their lack of database-

to-kernel knowledge transfer leaves kernel choices unconditioned on the selected database knobs, supporting the benefit of the functional context used by MixTuner.

Measurement aggregation. The three repetitions cover all methods and all four database–workload pairs in Table 1, including Reference Config. Each repetition measures TPC-C throughput or TPC-H P95 latency using the metric definition in Section 4.1. Each table entry gives the arithmetic mean of the three measurements, followed by their standard deviation in parentheses as $\pm$ standard deviation, in the same units as the corresponding metric. For TPC-H, both statistics are computed across the three per-run P95 latencies. The run details below document the original MySQL TPC-H records preceding these repetitions: GPTuner and Lasso-Rank Pruning provide search-best observations, while AutoOS provides a single benchmark measurement.

GPTuner on MySQL TPC-H. The original GPTuner run minimizes BenchBase P95 request latency, recorded in microseconds. With optimizer seed 100, it performs coarse and fine search. Fine search imports all coarse observations without repeating their workload executions. All recorded evaluations succeed. The final incumbent is configuration 54 in the fine-stage history, with a recorded P95 of 4,995,152 μs, rounded to 5.00 s. This original search-best observation agrees between the run history and the final incumbent record. Using the displayed values in Table 1, the P95 reduction relative to GPTuner is $(1 - 4.87/5.00) \times 100\% = 2.60\%$.

Lasso-Rank Pruning on MySQL TPC-H. Table 1 reports a mean P95 of 5.03 ± 0.67 s over three repetitions. The original September 24 search uses the distribution kernel `6.8.0-136-generic` and includes coarse and fine stages. Three recorded attempts fail and contribute penalty costs rather than performance values. Fine search imports five coarse observations into its history. Evaluation 68 in that earlier search attains a P95 of 5.13 s; this single-search observation agrees between the evaluation log, the saved fine-search result, and the corresponding BenchBase summary. The summary records MySQL 8.4.11, SF=1, $T = 6$, and serializable isolation. Earlier files for SF=0.1 and $T = 1$ in the same directory are outside this search and are excluded.

AutoOS on MySQL TPC-H. The September 24 BenchBase run records MySQL 8.4.11, SF=1, $T = 6$, serializable isolation, 44 measured requests, and P95 latency of 148.32 s. Its workload XML matches Reference Config in batch size and query-phase definitions. The database parameter snapshots agree except for three runtime fields: `pseudo_thread_id`, `statement_id`, and `timestamp`. This original AutoOS record therefore contains one measurement with the reference database settings. The supplied record contains no kernel-build artifact or kernel-search trajectory.

B.8 Tuning Time and API Cost

We summarize the recorded tuning time and LLM API expenditure of GPTuner, SysInsight, AutoOS + GPTuner, AutoOS + SysInsight, and MixTuner on MySQL/TPC-H at SF=1 and $T = 6$. Table 10 separates kernel tuning and database tuning time, reports their sum as total tuning time, and lists API expenditure in US dollars. Performance is reported in Table 1.

Table 10: Tuning time and LLM API cost for MySQL/TPC-H at SF=1 and $T = 6$. Total time is the sum of kernel and database tuning time. Performance results are reported in Table 1.

Method	Kernel tuning (min)	DB tuning (min)	Total time (min) ↓	LLM API cost (USD) ↓
GPTuner	-	632.3	632.3	2.20
SysInsight	-	19.8	19.8	-
AutoOS + GPTuner	227.0	632.3	859.3	4.50
AutoOS + SysInsight	227.0	19.8	246.8	2.30
MixTuner	151.4	171.4	322.8	3.90

Time and monetary cost. MixTuner requires 322.8 minutes in total, comprising 151.4 minutes of kernel tuning and 171.4 minutes of database tuning. This is 48.95% less time than GPTuner and 62.43% less than AutoOS + GPTuner. Its LLM API cost is $3.90, compared with $2.20 for GPTuner

and $4.50 for AutoOS + GPTuner. SysInsight and AutoOS + SysInsight finish sooner, at 19.8 and 246.8 minutes, respectively; AutoOS + SysInsight also has a lower API cost of $2.30.

Performance–cost tradeoff. Table 1 shows that MixTuner achieves the best mean performance on all four tasks. On MySQL/TPC-H, its P95 latency is 4.87 ± 0.49 s, compared with 5.00 ± 0.60 s for GPTuner and 5.08 ± 0.56 s for AutoOS + GPTuner. Thus, MixTuner combines lower mean P95 latency with less tuning time than these two methods. SysInsight and AutoOS + SysInsight achieve 5.08 ± 0.61 s and 5.21 ± 0.57 s, respectively. Relative to these faster methods, MixTuner reduces mean P95 latency by 4.13% and 6.53%, at the cost of additional tuning time. These results characterize the tradeoff between tuning overhead and the performance of the resulting configuration on MySQL/TPC-H.

C LLM Prompt Templates

The following templates specify the inputs and output schemas used by the selection and configuration utilities. MySQL examples illustrate the format; the database type and task context are supplied by the caller. Instructions in a prompt express the intended behavior of the LLM, not a guarantee that its output is correct.

C.1 Database Knowledge Construction

This template extracts structured metadata from a MySQL configuration description. The parser also handles responses wrapped in a JSON code block.

```
You are a MySQL Database Configuration Expert and an automated parser.
Your goal is to analyze a MySQL configuration item based on its name and
official documentation description.

You must output a raw JSON object. Do not include markdown formatting.

### Classification Types
1. **NUMERIC**: Integers, floating-point numbers, or memory sizes.
2. **BOOLEAN**: Toggles like ON/OFF, 1/0.
3. **PATH**: File/Directory paths.
4. **ENUM**: Single selection from a predefined list.
5. **SET**: Multiple selections from a predefined list.
6. **STRING**: Free-form text, IP addresses, timestamps.

### Output JSON Schema
{
  "name": "Config Item Name",
  "type": "One of [NUMERIC, BOOLEAN, PATH, ENUM, SET, STRING]",
  "default_value": "Extracted value as string. If dynamic/unknown,
  return null",
  "is_dynamic": true or false,
  "numeric_min": 0,   // Number (int/float) or null. ONLY for NUMERIC.
  "numeric_max": 100, // Number (int/float) or null. ONLY for NUMERIC.
  "allowed_values": [] // Array of strings for ENUM/SET. Null for others.
}

### Critical Rules for NUMERIC Types
1. **Pure Numbers Only**: Fields `numeric_min` and `numeric_max` must be
  raw
JSON numbers (e.g., 1024, 0.5). Do NOT use strings.
2. **No Units/Text**: Remove "bytes", "seconds", "(64-bit platforms)".
  Convert
1G to 1073741824.
3. **Calculate Expressions**: If the doc says "2^64-1", calculate it and
  output
`18446744073709551615`. If it says "unlimited" or "infinity", set
  `numeric_max`
to `null`.
```

```
4. **Platform Ambiguity**: If the range differs by platform (32-bit vs
  64-bit),
**ALWAYS output the 64-bit value**.
5. **Format**:
   - If description is "0 to 65535", output `"numeric_min": 0,
   "numeric_max": 65535`.
   - If description is "Minimum is 1024", output `"numeric_min": 1024,
   "numeric_max": null`.
6. **Inference**:
   - **Implied Minimum**: If the minimum is not explicitly stated in the
  text,
    but the item represents a **size, count, timeout, or duration**,
  assume
    the `numeric_min` is **0**.
    - **Implied Maximum**: If the maximum is not stated, return `null`.
    - **Unsigned Assumption**: If the documentation mentions "unsigned" or
    implies positive values, set `numeric_min` to 0 unless otherwise
  specified.

### General Rules
- `is_dynamic`: true if "Dynamic: Yes" or "SET GLOBAL" is implied;
  otherwise false.
- For ENUM/SET, put options in `allowed_values` list: `["ON", "OFF"]`.
```

C.2 KNOB SELECTION

The LLM evaluates one functional group c at a time using the template below. An empty selection excludes that group from $S(g)$; otherwise its selected knobs contribute entries to $C(g)$ and $\mathcal{R}_{\mathrm{db}}(g)$. The returned names identify records in $\mathcal{T}_{\mathrm{db}}$, from which we attach the functional descriptions and documented configuration relationships that form $\mathrm{info}(x_i^c)$.

```
# Role
You are a Senior MySQL Database Performance Tuning Expert. Your task is
  to analyze
a given list of MySQL configuration parameters (knobs) and identify
  which ones are
critical for achieving a specific optimization goal.

# Input Format
The input will consist of:
1. An **Optimization Goal** (e.g., "Reduce 95th percentile latency for
  TPC-H
workload").
2. A **Parameter Block** containing a category name and a list of
  parameters with
their current values (YAML-like structure).

# Criteria for Selection
Select a parameter ONLY if:
1. Tuning it has a **significant probability** of positively impacting
  the
specific Optimization Goal.
2. It is generally considered a tunable "knob" (not a read-only system
  variable
or version number).

If the entire category contains no impactful parameters for the specific
  goal,
return an empty list.

# Output Format
Return **ONLY** a raw JSON list of strings containing the names of the
selected parameters. Do not include markdown code blocks (```json),
explanations, or any other text.
```

```
# Examples

## Example 1
**Optimization Goal:** Reduce 95th percentile latency for TPC-H (OLAP)
  workload
**Parameter Block:**
- InnoDB Settings
    - innodb_buffer_pool_size: 134217728
    - innodb_read_only: OFF
    - innodb_io_capacity: 200
    - innodb_version: 8.0.26

**Output:**
["innodb_buffer_pool_size", "innodb_io_capacity"]

## Example 2
**Optimization Goal:** Reduce Disk I/O usage
**Parameter Block:**
- Localization
    - lc_time_names: en_US
    - character_set_server: utf8mb4
    - collation_server: utf8mb4_general_ci

**Output:**
[]

## Example 3
**Optimization Goal:** Maximize throughput for write-heavy OLTP
**Parameter Block:**
- Performance Schema
    - performance_schema: ON
    - performance_schema_accounts_size: -1
    - performance-schema-consumer-events-stages-current: ON

**Output:**
["performance_schema"]

# Current Task

**Optimization Goal:** {{USER_GOAL}}
**Parameter Block:**
{{PARAMETER_BLOCK_CONTENT}}

**Output:**
```

C.3 Numerical Range Suggestions

For each numerical knob selected under goal g, the LLM suggests bounds using its documented metadata and hardware information. These bounds, together with the legal choices for Boolean and enumeration knobs, define the ranges recorded in $\mathcal{R}_{\mathrm{db}}(g)$. The numerical range prompt is:

```
# Role
You are an expert Database Administrator (DBA) specializing in MySQL
  performance
tuning and automated configuration optimization.

# Objective
Your goal is to significantly narrow down the search space (min/max
  range) for a
specific MySQL configuration knob. You must consider the hardware
  constraints
and the user's specific optimization goal.
```

```
# Guidelines for Reasoning
1. **Hardware Awareness:** precise calculations based on Total RAM and
  CPU Cores
are required. (e.g., specific percentage of RAM for buffer pools).
2. **Workload Awareness:** Distinguish between OLAP (e.g., TPC-H,
  complex queries)
and OLTP (e.g., Sysbench/TPC-C, high concurrency).
3. **Safety:** Ensure the recommended upper bound does not cause
  Out-Of-Memory
(OOM) errors or system instability.
4. **Aggressiveness:** The user wants the *smallest possible search
  range*. If a
heuristic exists (e.g., "normally set to 70% of RAM"), center the range
  around
that value with a small margin (e.g., +/-5% or +/-10%), rather than
  giving a safe
but wide default range.

# Output Format
You must output a raw JSON object strictly adhering to this schema, with
  no
markdown formatting or conversational filler:
{
    "reasoning": "A concise explanation of the calculation based on
  hardware
    and workload.",
    "suggested_min": <number>,
    "suggested_max": <number>
}
```

C.4 KERNEL MENU PRUNING

The `knobs_formatted` field supplies $C(g)$, grouping the selected knob information by functional category. Each knob entry includes its name, functional description, and documented relationships with other settings.

```
You are a Linux kernel and database tuning expert.
Your task is to identify which kernel configuration menu sections are
  IRRELEVANT
to the given database tuning goal.

You understand both:
1. How Linux kernel subsystems affect database performance (I/O
  scheduling,
memory management, networking, etc.)
2. How database parameters interact with kernel configurations

Be conservative: if a menu might be relevant, keep it. Only prune menus
  that
are clearly unrelated.

Respond ONLY in valid JSON format with these fields:
{
    "prune": ["menu_name1", "menu_name2", ...],  // Menus to prune
        (clearly irrelevant)
    "keep": ["menu_name1", "menu_name2", ...],   // Menus to keep
        (potentially relevant)
    "reasoning": {
        "menu_name": "reason why prune/keep"
    }
}

## Tuning Goal
{goal}
```

```
## Database Type
{self.db_type.upper()}

## Database Knob Information (organized by functional group)
{knobs_formatted}

## Current Kernel Configuration Scope
The following are the menu items in the current kernel configuration
  scope:
{menu_list}

## Task
Analyze each menu item and decide:
1. Which menus are CLEARLY IRRELEVANT to the database tuning goal and can
be pruned?
2. Which menus should be KEPT for further exploration?

Consider these factors:
- Memory management menus are usually relevant to database performance
- I/O and block device menus are relevant to storage-bound workloads
- Network menus are relevant if the database serves remote clients
- CPU scheduling menus affect query execution
- Filesystem menus affect data durability and I/O patterns
- Security menus may affect performance overhead
- Hardware-specific drivers (sound, graphics, etc.) are usually
  irrelevant

Provide your analysis in JSON format.
```

C.5 KERNEL OPTION ASSIGNMENT

The database-aware assignment routine uses the following system prompt, together with the goal, database type, grouped knob information $C(g)$, and current option metadata from $\mathcal{T}'_{\text{ker}}$.

```
You are a Linux kernel tuning expert optimizing for database workloads.
Your task is to suggest kernel configuration changes that will improve
  database performance.

You understand:
1. How kernel parameters affect I/O, memory, CPU scheduling, and
  networking
2. How these kernel settings interact with database internals
3. Trade-offs between different configuration choices

For each option, decide whether to:
- CHANGE: Modify the value to optimize for the goal
- KEEP: Keep the current value (it's already optimal or safe default)
- SKIP: This option is not relevant enough to change

Respond ONLY in valid JSON format:
{
    "changes": [
        {
            "name": "CONFIG_NAME",
            "current_value": "n",
            "new_value": "y",
            "reason": "Brief explanation"
        }
    ],
    "keep": [
        {
            "name": "CONFIG_NAME",
            "reason": "Why keeping current value"
```

```
        }
    ],
    "summary": "Overall summary of changes"
}

IMPORTANT GUIDELINES:
1. For BOOL options: use "y" (enable) or "n" (disable)
2. For TRISTATE options: use "y", "m" (module), or "n"
3. For INT/HEX options: provide appropriate numeric values
4. Be conservative - only change what clearly benefits the goal
5. Consider dependencies - some options may affect others
```

D Discussion

MixTuner connects database configuration knowledge to kernel configuration decisions, then uses workload measurements to select database values on the resulting kernel. This interface and execution order define the approach. The experiments assess the configurations it produces under the recorded workloads and search budgets. Below, we relate these design choices to the interpretation of the results and to deployment in other environments.

D.1 Context Transfer and Conditional Search

The shared goal g specifies what to optimize, while $C(g)$ describes the database functions selected for tuning. Supplying this context during both menu filtering and option assignment lets kernel decisions depend on the database knobs under consideration. Functional descriptions provide a common interface across differently organized configuration hierarchies. They express hypotheses about useful kernel support; Kconfig rules and workload execution determine whether the resulting configuration can be used. The method combines this interface with existing tools, including Kconfiglib, SMAC3 (Lindauer et al., 2022), and coarse-to-fine database search (Lao et al., 2024). The methodological contribution concerns how information and execution are coordinated across the two configuration layers.

Co-tuning in Equation 1 permits both database and kernel settings to change. MixTuner makes those decisions in sequence: selected database-knob context conditions kernel generation, and the deployed kernel conditions database value search. Within a run, measurements update the database optimizer while the kernel and context remain fixed. Thus, the joint objective in Equation 2 is a search target, with Algorithm 1 providing a budgeted procedure for approaching it. The procedure can miss useful kernel configurations or exclude useful database values during range selection. A larger admissible configuration space alone gives no guarantee that a finite search will outperform a single-layer tuner. Revisiting kernel choices after database measurements would explore this tradeoff further, at the cost of additional kernel transitions.

D.2 Feasibility and Deployment Cost

Preparing a kernel once allows many database trials to share compilation, boot validation, and deployment. This is the computational reason for the staged schedule. Appendix B.8 summarizes the recorded tuning time and LLM API expenditure of five methods on MySQL/TPC-H at SF=1 and $T = 6$. MixTuner uses less total tuning time than GPTuner and AutoOS + GPTuner, while SysInsight and AutoOS + SysInsight finish sooner. Together with the performance results in Table 1, these measurements quantify the performance–cost tradeoff in this setting.

Kconfig legality, successful compilation, bootability, and workload execution are distinct validation conditions. For example, a kernel capability can support a database feature while its usable capacity still depends on runtime resource provisioning (Appendix A.5). The database search penalizes failed trials and returns a successfully measured configuration, or reports failure when none exists. These checks establish execution under the benchmark conditions. Deployment in a different environment also requires validation of resource limits, durability, recovery, and workload behavior. Those properties cannot be inferred from a throughput or P95 measurement alone.

D.3 Reading the Performance Comparisons

The 36-scenario matrix measures the effect of applying the complete workflow relative to the recorded untuned Reference Config. Its 35 improvements and one regression describe the observed outcomes across database systems, workloads, scale factors, and terminal counts. Each ratio compares the same scenario. For TPC-H, the inverse P95 ratio measures a reduction in per-run tail latency for the configured query execution; it is not an estimate of query throughput. The differing MySQL and PostgreSQL workload phases in Appendix B.1 also matter when interpreting their absolute latencies.

The 36-scenario evaluation and the configuration-layer and base-model ablations combine search-best observations with the explicitly identified single measurements. Selecting the best observation can favor a configuration that benefited from measurement noise. These records therefore characterize observed search outcomes, without estimating the mean performance or variability of repeated searches and fixed-configuration deployments. The small base-model gaps require that distinction in particular. For these evaluations, independent repetitions, together with matched cache state, phase boundaries, and durability settings, are needed to establish their stability. Table 1 reports means and standard deviations over three repetitions for every method–task pair.

Reference Config comparisons measure gains over the untuned configuration, while Table 1 compares MixTuner with nine tuning baselines at SF=1 and $T = 6$. MixTuner achieves the best mean performance on all four tasks. Relative to the strongest baseline in each column, its TPC-C throughput gains are 2.18% on MySQL and 2.04% on PostgreSQL, and its TPC-H P95 reductions are 2.60% and 5.26%, respectively. AutoOS + GPTuner and AutoOS + SysInsight also tune both layers, but pass no database-knob functional context to kernel tuning. Their lower performance across all four tasks supports the value of cross-layer knowledge transfer beyond composing separate tuners. Appendix B.7 gives the comparison details. Initialization and total tuning cost also depend on the method.

D.4 Interpreting the Configuration Variants

Figure 4 compares complete tuning with workflows that retain a reference layer. Full MixTuner attains the best recorded outcome among these variants on all four tasks. The reference-kernel workflow reoptimizes database settings on that kernel, and the generated and reference kernels use different Linux builds. Consequently, this comparison includes kernel replacement and the database search it enables. Establishing the effect of one kernel option, or a database–kernel interaction for fixed configurations, requires corresponding controlled measurements with the same artifacts. Appendix B.4 specifies these distinctions and the unresolved identity of the Reference DB kernel.

The composed baselines in Table 1 test whether separate kernel and database tuners can match MixTuner without sharing database-knob functional context. Their results support coordinating the two layers through this context. Appendix B.5 further compares absent, flat, and grouped database context while retaining the selected knobs, search ranges, and optimizer settings. These variants assess context transfer and functional grouping. Separating the contributions of kernel-menu filtering and database-range refinement would require additional controlled variants within MixTuner.

D.5 Model Dependence and Transfer

The three tested base models produce similar search-best MySQL TPC-H latencies under the matched scenario in Table 2. This demonstrates operation with multiple backbones in that setting. Each model contributes one selected search, so the comparison leaves prompt sensitivity and variation across seeds, model versions, and other workloads unmeasured. Appendix B.6 identifies the selected runs. Table 3 in Appendix B.1 lists additional reproducibility records, including model snapshots and decoding settings.

The design separates configuration knowledge, language-model judgments, and workload measurement through explicit interfaces. Applying it to another database or platform requires suitable knob metadata, kernel knowledge, configuration adapters, and a benchmark objective that preserves the intended service semantics. Existing configurations provide starting points; performance must be remeasured in the target environment. Extending the evaluation to additional platforms and repeated runs would test how broadly the observed gains transfer.

E LIMITATION AND FUTURE WORK

Feedback across tuning stages. MixTuner fixes the database context and deployed kernel before optimizing database values. This schedule amortizes kernel preparation across database trials, but measurements cannot revise earlier kernel decisions. A kernel chosen from knob descriptions may therefore miss opportunities revealed by the eventual database configuration. Future work could use workload measurements and resource profiles to decide when to revisit kernel choices. A small pool of candidate kernels, with database search conditioned on each candidate, would provide one way to explore these interactions. The central question is whether the additional performance gain justifies compilation and deployment within a fixed total tuning budget.

Dependence on configuration knowledge and LLM judgments. Knob selection, numerical range suggestions, and kernel-menu filtering restrict the configurations that subsequent stages can examine. Incomplete documentation or an incorrect relevance judgment can exclude useful choices, and database optimization cannot recover a knob or kernel branch removed before measurement. Kconfig dependency checks constrain legal assignments but do not establish their performance value. A useful extension would make these restrictions revisable: reserve a small measurement budget for omitted knobs, expand numerical ranges when promising observations approach their boundaries, and revisit pruned menus when resource profiles indicate an unexplored bottleneck. Keeping configuration knowledge aligned with database and kernel versions would also support transfer to new environments.

Evaluation scope and variability. Our evaluation covers MySQL and PostgreSQL with TPC-C and TPC-H on one server platform. The 36 scenarios vary workload scale and terminal count, while comparisons with tuning baselines use SF=1 and $T = 6$. Table 1 reports three repetitions per method–task pair; the broader scenario matrix and base-model ablation include search-best observations and the single measurements described in Appendix D.3. These experiments leave variability across complete tuning runs and transfer across hardware platforms unresolved. Future evaluation should repeat the full search with independent seeds, remeasure selected configurations in separate runs, and extend the workload and hardware coverage. Controlled comparisons using the same kernel source version would further isolate the contribution of individual configuration choices.

Operating objectives and adaptation. The current formulation optimizes throughput or P95 latency for a fixed workload and holds runtime operating-system settings constant. Kernel preparation incurs an upfront cost, and the time and API-cost comparison in Appendix B.8 covers one MySQL/TPC-H setting. How much workload execution is needed to recover that cost remains an open practical question. Future work could evaluate the accumulated benefit over longer execution periods and extend tuning to changing workload mixes. Coordinating database knobs with runtime operating-system controls could provide a less costly adaptation path between kernel rebuilds, while explicit resource and service constraints would allow performance improvements to be assessed under deployment requirements.