Why Database Manuals Are Not Enough: Efficient and Reliable Configuration Tuning for DBMSs via Code-Driven LLM Agents

Xinyi Zhang
Renmin University, China
xinyizhang.info@ruc.edu.cn

Tiantian Chen
Renmin University, China
chenentt@ruc.edu.cn

Zhentao Han
Renmin University, China
avaleph@ruc.edu.cn

Zhaoyan Hong
Renmin University, China
hongzhaoyan@ruc.edu.cn

Wei Lu
Renmin University, China
lu-wei@ruc.edu.cn

Sheng Wang
Alibaba Group
sh.wang@alibaba-inc.com

Mo Sha
Alibaba Group
shamo.sm@alibaba-inc.com

Anni Wang
Alibaba Group
wanganni.wan@alibaba-inc.com

Shuang Liu
Renmin University, China
shuang.liu@ruc.edu.cn

Yakun Zhang
Peking University
zhangyakun@stu.pku.edu.cn

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

Xiaoyong Du
Renmin University, China
duyong@ruc.edu.cn

ABSTRACT

Modern database management systems (DBMSs) expose hundreds of configuration knobs that critically influence performance. Existing automated tuning methods either adopt a data-driven paradigm, which incurs substantial overhead from trial-and-error exploration, or rely on manual-driven heuristics extracted from database documentation, which are often limited (since not all knobs have documented guidance) and overly generic (thus not well-suited for specific workloads), leading to costly and unstable tuning performance. Motivated by the fact that the control logic of configuration knobs is inherently encoded in the DBMS source code, we argue that promising tuning strategies can be mined directly from the code, uncovering fine-grained insights grounded in system internals.

To this end, we propose `SysInsight`, a code-driven database tuning system that automatically extracts fine-grained tuning knowledge from DBMS source code to accelerate and stabilize the tuning process. `SysInsight` combines static code analysis with LLM-based reasoning to identify knob-controlled execution paths and extract semantic tuning insights. These insights are then transformed into quantitative and verifiable tuning rules via association rule mining grounded in tuning observations. During online tuning, function-level system diagnosis is applied to identify critical knobs, which are subsequently adjusted under the guidance of these rules. Extensive evaluations demonstrate that compared to the SOTA baseline, `SysInsight` converges to the best configuration on average 7.11× faster while achieving a 19.9% performance improvement, validating its tuning efficiency and reliability.

PVLDB Reference Format:
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. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Blairruc-pku/SysInsight.

1 INTRODUCTION

Modern database management systems (DBMSs) expose hundreds of configuration knobs [15], and setting suitable values for these knobs is crucial for achieving high throughput and low latency. However, determining the optimal configurations is an NP-hard problem [45]. Traditionally, database administrators (DBAs) invest significant effort in manually tuning these knobs for specific workloads. To alleviate this burden, many recent studies have focused on automating knob tuning using machine learning (ML) techniques.

Depending on the source of knowledge they utilize, we categorize existing ML-based tuning approaches into two broad paradigms: (1) *data-driven* methods, which interact iteratively with the DBMS to collect performance data under various configurations and adapt tuning decisions based on the feedback; and (2) *manual-driven* methods, which extract tuning guidance from database manuals, online technical blogs or Q&A forums. *Data-driven* methods [5, 11, 30, 36, 42–44, 51, 55] formulate tuning as a black-box optimization problem, which explores the search space via trial-and-error. In the absence of effective guidance to prune or prioritize configurations, these methods must iteratively probe the search space, leading to substantial overhead due to extensive workload replays to gather performance observations.

Manual-driven methods [31, 48] have recently been proposed to leverage pretrained language models to extract expert heuristics from textual documentation for database tuning. For example, GPTuner [31] uses large language models (LLMs) to extract tuning ranges and recommended values. However, the coverage of such heuristics is limited, as not all knobs have expert recommendations recorded in manuals. In GPTuner, for instance, fewer than 50% of the knobs are identified to have meaningful ranges smaller than their default ranges (Section 6.2). For newly developed or poorly documented systems, the tuning heuristics may be entirely absent, rendering *manual-driven* approaches ineffective. Second, the information extracted by existing methods is often oversimplified, which lacks fine-grained guidance. For example,

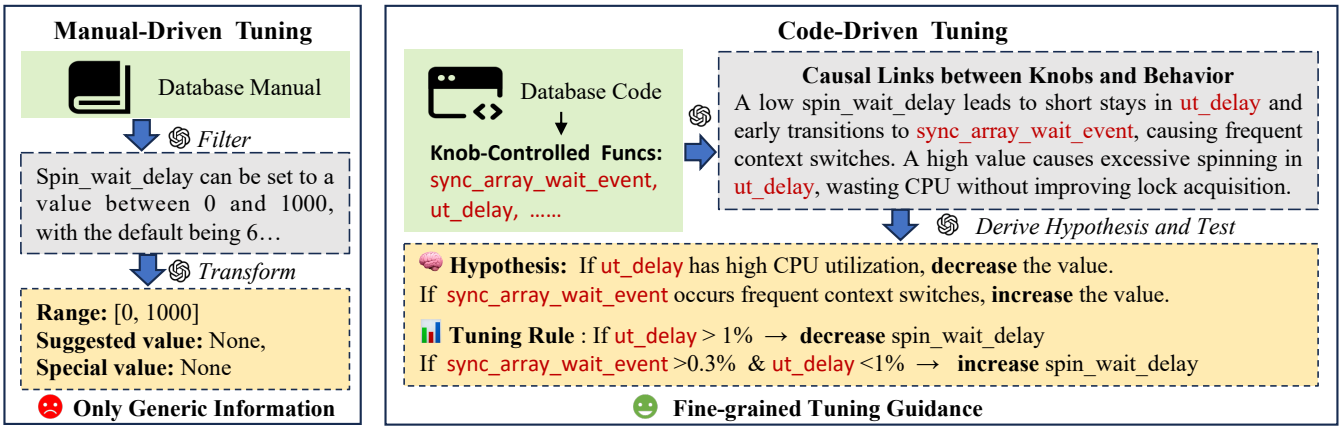


Figure 1: Motivating Example. Manual-driven approach only provides generic information for knob `spin_wait_delay` (i.e., `innodb_spin_wait_delay`), requiring further trial-and-error. `SysInsight` addresses this gap by analyzing the knob-controlled functions (e.g., ut_delay, sync_array_wait_event) to derive fine-grained tuning guidance rooted in system internals.

`innodb_buffer_pool_size` is suggested to be set to 70%–80% of the machine's physical memory [31, 48]. However, setting the optimal value for a particular workload still relies on trial-and-error within this range, and may even fall outside it, since these ranges are summarized for common scenarios rather than tailored to specific workloads (Section 6.4).

Motivation. Beyond relying on existing manuals, in this paper, we argue that tuning strategies can be derived from the dedicated system code itself, as configuration knobs fundamentally influence performance by controlling predefined execution paths (Section 2.2). From this perspective, we propose a new *code-driven* tuning paradigm, which we instantiate in a system called `SysInsight`, designed to act as a System architect with deep Insight into the database's internal mechanisms. `SysInsight` enables precise configuration tuning by identifying runtime performance bottlenecks, analyzing the relevant system code to locate the controlling knobs, and tuning them based on their impact on execution paths.

To illustrate how code-driven tuning differs from existing manual-driven methods, Figure 1 compares `SysInsight` with GPTuner [31], using `innodb_spin_wait_delay` as an example. GPTuner fails to obtain compact ranges (the 0–1000 range merely reflects MySQL's default bounds) or reliable recommended values due to the lack of expert heuristics and the absence of widely accepted tuning guidelines for `innodb_spin_wait_delay` available in the manual. In contrast, `SysInsight` analyzes the database source code to identify the functions affected by `innodb_spin_wait_delay`, finding its control over the trade-off between `ut_delay` and `sync_array_wait_event`. Based on these causal links, `SysInsight` formulates hypotheses and refines them into actionable tuning rules through empirical validation, uncovering fine-grained tuning guidance beyond what is provided in database manuals.

There are several challenges in achieving automatic *code-driven* tuning. First, while the effects of knobs are encoded in the DBMS codebase, analyzing their control logic and associated behavioral patterns is highly complex and labor-intensive. Recent advances in LLMs have demonstrated strong capabilities in code generation and reasoning [25, 39]. However, modern DBMS codebases often exceed millions of lines, making it difficult to isolate how a knob influences execution (e.g., which functions or loops it controls) without overwhelming the LLM or losing context. Moreover, existing code summarization techniques [46] are primarily designed for functional documentation rather than identifying knob-specific control paths. Thus, a challenge lies in automatically distilling tuning-relevant insights such as high-level control dependencies and performance implications from the large and complex codebase (**C1**).

Second, even when such insights are available, converting them into *reliable tuning guidance* remains difficult (**C2**). With hundreds of knobs, it is often unclear which ones are more likely to yield performance improvements. Even for a promising knob, determining the adjustment direction (increase or decrease) and step size is nontrivial. For example, although we may know that a high `ut_delay` suggests decreasing `innodb_spin_wait_delay`, this insight is insufficient without defining a safe range and precise conditions. Effective guidance must specify both the triggering conditions and the adjustment to avoid aggressive changes that might downgrade system performance; for instance, "if the `ut_delay` ratio exceeds a certain threshold, decrease `innodb_spin_wait_delay` by a certain step until the condition is no longer met". Crafting such guidance is difficult as it requires precisely specifying conditions, directions, and steps under diverse contexts without introducing instability.

Third, the absence of a robustness verification mechanism presents a critical challenge, particularly for mitigating issues such as hallucinations in LLMs [23]. Unlike general-purpose text generation, knob tuning operates in high-stakes environments where each recommendation can directly impact throughput, latency, or resource contention. However, existing approaches—whether data-driven or manual-driven—often explore a wide range of suboptimal or even harmful configurations during tuning, as they lack safeguards to assess the trustworthiness of the tuning logic beforehand (Section 6.3). To ensure safety and effectiveness, tuning decisions must be grounded in real system behavior and verifiable prior to deployment. Yet both data-driven models and LLMs offer no intrinsic guarantees of correctness or generalizability, and tuning outcomes

are highly context-dependent. As a result, assessing whether a tuning knowledge is reliable remains an open and difficult problem, yet essential for trustworthy tuning outcomes (**C3**).

`SysInsight` addresses these challenges with the following designs. First, to uncover knob effect from large-scale codebase, we propose a code-driven tuning knowledge extraction approach that combines static analysis and LLM-based code reasoning. Static analysis localizes functions affected by target knobs to reduce irrelevant code exploration, while an LLM agent explores their surrounding context (e.g., callees and class definitions) to formulate potential tuning hypotheses based on their impact on system behavior (for **C1**). Second, to transform these semantic tuning hypotheses into effective and reliable tuning guidance, we introduce a rule induction framework that uncovers knob adjustments consistently associated with performance improvements under similar runtime contexts (for **C2**). Third, we propose a reliability verification mechanism that maintains rule-level confidence scores based on performance feedback, which supports continuous rule refinement and safe reuse (for **C3**). Finally, we integrate these components into a code-driven, rule-augmented tuning framework which identifies bottleneck functions, retrieves relevant rules, and suggests adjustments for the associated knobs. The main contributions are summarized as follows:

- We leverage database source code to trace knob-to-function influence paths, identify critical knobs, and uncover promising tuning strategies. To the best of our knowledge, this work presents the first automatic code-driven configuration tuning paradigm.
- We define context-aware and verifiable tuning rules and develop a customized association rule mining pipeline that transforms promising tuning strategies into reliable, quantitative adjustments grounded in empirical observations.
- We integrate code-driven extraction and rule induction into a unified framework that adaptively retrieves, verifies, and refines tuning rules, enabling efficient and reliable knob tuning.
- We conduct extensive experiments to validate the effectiveness of `SysInsight`, which on average converges to the best configuration 7.11× faster, while achieving a 19.9% performance improvement compared to the second-best baseline.

2 PRELIMINARY

2.1 Database Knob Tuning

Consider a database system with a set of configuration knobs denoted by $\{\theta_1, \ldots, \theta_m\}$. We define its configuration space as $\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_m$, where each Θ_i denotes the domain of knob θ_i, which can be either continuous or categorical. We denote the database context metrics as $C = \{\mathcal{R}, \mathcal{W}\}$, where $\mathcal{R}$ represents runtime metrics (e.g., function sampling rates, internal counters), and $\mathcal{W}$ denotes workload characteristics. Let f be the performance metric to optimize. Given a specific configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ and context metrics $c \in C$, the corresponding performance $f(c, \boldsymbol{\theta})$ can be observed after evaluation in the database system. Assuming the objective is a maximization problem, the goal of database knob tuning is to identify an optimal configuration $\boldsymbol{\theta}^* \in \boldsymbol{\Theta}$ that maximizes performance under context c, i.e.,

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} f(c, \boldsymbol{\theta}). \tag{1}$$

To tackle this problem, existing efforts mainly focus on two dimensions: knob selection and configuration recommendation:

Knob Selection. Modern database systems expose hundreds of knobs, but their performance impact can vary significantly across different workloads. The goal of knob selection is to identify which knobs need to be tuned for the current workload. We discuss existing approaches from the two perspectives:

- *Data-driven* methods [6, 9, 18, 56] sample a large number of observations under different configurations and fit a ML model based on the observations to rank the knobs in terms of their importance measurement [38, 40, 47]. However, prior studies [26] have shown that accurately identifying important knobs requires substantial sampling cost, making it inefficient in practice.
- *Manual-driven* methods [31, 34] rely on LLMs to suggest important knobs. These methods prompt the LLM with workload descriptions (e.g., OLTP vs. OLAP) and query plans, assuming that its pre-training on manuals and forum discussions enables effective recommendations. However, such methods tend to favor popular knobs frequently seen in the pre-trained data of LLMs while overlooking critical but under-documented ones.

Configuration Recommendation. Once the tuning knobs are selected, the next step is to recommend their configuration values.

- *Data-driven* methods works iteratively: applies a suggested configuration to the DBMS, and update their internal model or strategy based on the observed performance. Depending on the underlying strategy, these methods can be further categorized into: (1) Heuristic-based methods [2, 8, 41, 59], which rely on rule-based or search-based heuristics to explore configuration space; (2) Bayesian Optimization (BO)-based methods [6, 10, 10, 16, 27, 55, 56], which model the configuration–performance relationship using a probabilistic surrogate and select the next configuration by maximizing an acquisition function; (3) Reinforcement Learning (RL)-based methods [9, 19, 33, 49, 52], which treat tuning as a sequential decision-making process and learn policies through interactions with DBMSs.
- *Manual-driven* methods [31, 48] further accelerate configuration search by extracting expert knowledge from manuals or online content to prune the search space. For example, DB-BERT [48] transforms textual descriptions into tuning hints of the form $P = f(v, S)$, where v is a candidate configuration, S encodes hardware properties (e.g., memory size) and P is the suggested value given v and S (e.g., 70% of the instance memory). However, these recommendations typically consider only hardware factors and do not adapt to workload characteristics, requiring further RL-based exploration to refine configurations. GPTuner [31] extracts min/max bounds, default, suggested, and special values for each knob and then applies BO to explore a reduced and discretized search space defined by these ranges and values.

2.2 Knob Impacts on DBMS Behaviors

We analyze how knobs influence DBMS behaviors by controlling the execution paths defined in the system code, and further analyze how this influence can be leveraged for performance tuning.

Explicit Control: Configuration knobs are commonly used as switches in software, enabling users to adjust various functional features of the system. In DBMS, knobs often appear in conditional

```
uint64_t log_reserve_and_open(uint len_upper_limit) {
  len_upper_limit = LOG_BUF_WRITE_MARGIN + (5 * len) / 4;

  if (log->buf_free + len_upper_limit > log->buf_size) {
    log_buffer_flush_to_disk();
  }
}
```

(a) Explicit Control-Dependent on `innodb_log_buffer_size`.

```
void rw_lock_x_lock_wait_func(rw_lock_t *lock) {
 while (lock->lock_word < threshold) {
  if (srv_spin_wait_delay) {
   ut_delay(random(0, srv_spin_wait_delay));
  }
  sync_array_wait_event();
 }
```

(b) Implicit Control-Dependent on `innodb_spin_wait_delay`.

Figure 2: Simplified Code Snippets Showing How Knobs Influence System Behavior: (a) `innodb_log_buffer_size`, implemented internally via `log->buf_size`, determines whether a log buffer flush is triggered. (b) `innodb_spin_wait_delay`, implemented internally via `srv_spin_wait_delay`, controls the polling frequency in the spin-lock loop.

statements (e.g., if, switch, for, and while), and determine which branch of the program will be executed. As a result, the dominated branches and blocks are control-dependent on the knob.

Example 2.1. `innodb_log_buffer_size` determines the size of the buffer that InnoDB uses to write to the log files on disk. As shown in Figure 2a, if the `buf_size` is smaller than the free size plus the length of new log, MySQL will trigger a costly synchronous buffer flush operation. This is an explicit control. If there has transactions with large blob/text fields, the buffer can fill up very quickly and incur performance hit.

Implicit Control. Beyond explicit control flow defined by conditional branches, configuration knobs can also implicitly propagate control dependency. For instance, if a knob influences the parameter of a delay operation (e.g., a `sleep` call) within a loop, then all statements inside the loop are implicitly control-dependent on that knob. This is because the knob alters the execution frequency and timing, thereby indirectly affecting control behavior.

Example 2.2. As shown in Figure 2b, when `srv_spin_wait_delay` is non-zero, `ut_delay()` injects a random delay (from 0 to `srv_spin_wait_delay`) before rechecking the lock condition. A larger `innodb_spin_wait_delay` value increases the delay in each iteration, reducing the frequency of calls to `sync_array_wait_event()`. Therefore, all basic blocks inside the `while` loop are implicitly control-dependent on `innodb_spin_wait_delay`.

Tuning Hypothesis. After analyzing the impacts of a knob on DBMS behaviors, one can formulate its potential tuning strategies, such as adjustment directions given inferred from the observed state of the associated functions. These strategies are promising but not yet verified. Thus, we refer to them as tuning hypotheses. A tuning hypothesis semantically provides an initial adjustment direction based on profiling signals of the functions a knob controls.

Example 2.3. Given the control effect of `innodb_log_buffer_size`, one can hypothesize that frequent execution of `log_buffer_flush_to_disk()` suggests increasing `innodb_log_buffer_size`. Similarly, given the control effect of `innodb_spin_wait_delay`, one can hypothesize that: If `ut_delay()` consumes excessive CPU time, lowering `innodb_spin_wait_delay` is recommended to reduce spin overhead; conversely, if `sync_array_wait_event()` dominates, increasing the delay helps avoid frequent context switches.

Although the tuning hypotheses are formulated on a per-knob basis, the presence of co-triggered bottleneck functions can implicitly reflect inter-knob dependencies. For example, frequent invocation of `buf_LRU_get_free_block` and `buf_flush_sync_all_buf_pools` indicate pressure on both memory and log subsystems, suggesting that both `innodb_buffer_pool_size` and `innodb_log_file_size` should be enlarged.

3 SYSTEM OVERVIEW

`SysInsight` emulates the action of a DBMS expert who identifies system issues, examines related code, performs targeted tuning, and summarizes tuning experience for future reuse. Figure **??** illustrates the workflow of `SysInsight`, which consists of both online and offline phases. The offline phase builds a reliable repository of code-driven tuning knowledge (Section 4). During online tuning, bottleneck functions are identified and their associated knobs are tuned under the guidance of the learned knowledge (Section 5).

Code-Driven Hypothesis Formulation. This process leverages the extracted control and data flow to trace how the knob's value propagates through the program logic and impacts system execution. A code retrieval agent gathers relevant unresolved context via Abstract Syntax Tree (AST) search. Based on the context, a summary agent formulates semantic tuning hypotheses expected to improve system performance.

Experimentation and Rule Derivation. Based on the semantic tuning hypotheses, `SysInsight` conducts tuning experiments by sampling configurations that conform to these hypotheses. To further distill precise and actionable tuning guidance from the experimental observations, association rule mining techniques are applied to extract tuning rules that explicitly specify the conditions under which knob adjustments should be triggered, as well as the direction and step size of each adjustment.

System Diagnosis and Knob Selection. During tuning, `SysInsight` collects runtime metrics and detects the deviations from expected performance to locate bottleneck functions. To identify relevant configuration knobs, `SysInsight` then performs static analysis on the DBMS source code, tracing the control and data flows to construct function–knob mappings, allowing `SysInsight` to pinpoint the knobs that directly or indirectly affect the identified bottlenecks.

Rule-Augmented Configuration Recommendation. `SysInsight` leverages the derived tuning rules in conjunction with semantic hypotheses to guide online configuration tuning. It retrieves relevant rules whose antecedent predicates match the current runtime metrics, ranking them by expected improvement to prioritize the most promising candidates and accordingly suggests a configuration. After applying the suggested configuration, `SysInsight` observes the

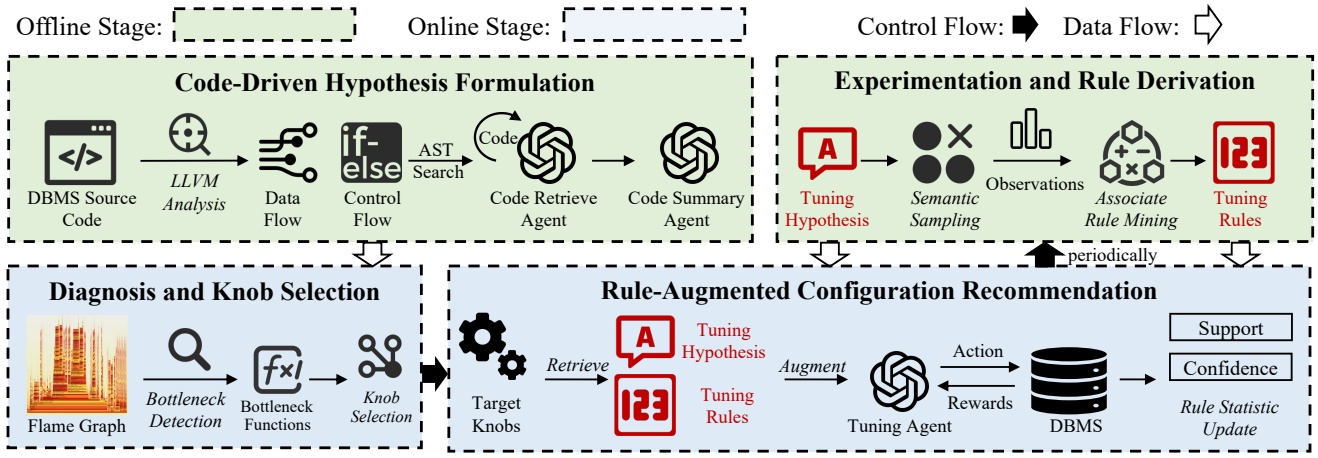


Figure 3: `SysInsight` Workflow.

resulting performance and updates rule statistics to track the reliability of each rule. As tuning observations accumulate, `SysInsight` periodically applies the association rule mining to induce tuning rules.

4 CODE-DRIVEN KNOWLEDGE HANDLER

4.1 Function-Knob Mapping Construction

To accurately determine how configuration knobs influence system execution while avoiding excessive input to the LLM, `SysInsight` first constructs a function–knob mapping that identifies the functions controlled by each knob. This mapping is essential for linking configuration changes to concrete execution paths and for guiding subsequent hypothesis formulation and knob selection.

Given a target knob k, `SysInsight` first leverages ConfMapper [58] to locate the initial program variable v^k to which the knob configuration is mapped to. Then, it adopts static taint analysis technique [50] to track both data-flow and control-flow propagation from v^k by analyzing the LLVM Intermediate Representation (IR) [32], which helps to determine the set of affected functions.

Data-Flow Analysis: Variables propagate through operation statements (e.g., assignment or arithmetic), function parameters or return values. In addition to explicit data-flow propagation, knob configuration can also affect variables implicitly through conditional statements. Given the initial variables v^k, `SysInsight` performs data-flow analysis to identify the set of related variables (denoted as $\widetilde{V}^k$) propagated by v^k.

Control-Flow Analysis: Based on $\widetilde{V}^k$, `SysInsight` further identifies functions that are control-dependent on these variables. The resulting set of controlled functions, denoted as F^k, serves as the basis for subsequent hypothesis formulation.

Example 4.1. Given the target knob `innodb_log_buffer_size`, `SysInsight` first locates the initial program variable defined by the `Sys_var_*` data structures in the file `sys_vars.cc`. It then performs data-flow analysis to identify related variables, such as `log->buf_size`, as shown in Figure 2a. Finally, `SysInsight` identifies the functions that are control-dependent on these variables, such as `log_buffer_flush_to_disk()`, which are triggered when specific conditions involving the configuration knob are met.

After analyzing all knobs, `SysInsight` builds function–knob association mappings that record, for each function f, the set of controlling knobs K^f such that adjusting any $k \in K^f$ influences the execution behavior of f. For instance, the execution of `log_buffer_flush_to_disk()` is governed by `innodb_log_buffer_size`, while `buf_read_ahead_random()` is controlled by both `innodb_buffer_pool_instances` and `innodb_random_read_ahead`. This mapping allows `SysInsight` to identify knobs that control the performance-bottleneck functions, enabling more targeted tuning.

4.2 Code-Driven Hypothesis Formulation

Building upon the previously extracted data and control flows, `SysInsight` utilizes LLMs to analyze and summarize the influence of knob k on the execution paths F^k via iterative collaboration between a code retrieval agent and a code summary agent. Much like how developers read and reason about source code, understanding the effect of a knob often requires navigating across multiple files and symbols to resolve API calls and dependencies. To enable this reasoning, `SysInsight` parses the database codebase into an abstract syntax tree (AST) and provides two structured search APIs to facilitate efficient and targeted retrieval of relevant code fragments: (1) `search_function`(f) returns the implementation of the specified function; (2) `search_class`(cls) returns the signature of the specified class.

The retrieval process begins with the parent function that contains the control logic linking knob k to the controlled function(s) in F^k. From this context, the retrieval agent is guided to resolve any missing function or class references whose definitions are essential for deepening the understanding of how the knob influences system behavior. To support this process, `SysInsight` adopts a stratified context search strategy [57], which iteratively prompts the retrieval agent to determine which API calls are needed based on the current context. After the API invocations in a given stratum are executed, the newly retrieved code snippets are incorporated into the context. The retrieval agent is then instructed to assess

whether the context is sufficient to analyze the influence of the configuration knob. Based on this assessment, the retrieval agent decides whether to (a) continue the iterative search process or (b) terminate retrieval when the code context is deemed complete.

Example 4.2. To understand how `innodb_spin_wait_delay` affects F^k, the retrieval agent first receives the code snippet of the parent function `rw_lock_x_lock_wait_func()` and is prompted to assess whether the context is sufficient. In the first stratum, it invokes the `search_function` API to retrieve the implementations of unresolved functions within `rw_lock_x_lock_wait_func()`, which are crucial for understanding the knob's influence, such as `sync_array_wait_event()`. In the second stratum, the context is augmented with the retrieved results. At this stage, the retrieval agent assesses that the context is sufficient and terminates the search process.

Once the retrieval agent determines that the context is sufficient or the query budget has been exhausted, the collected context is passed to the summary agent, which generates a structured reasoning chain that explains how the knob influences its associated functions and impacts overall database performance. Specifically, to understand the underlying causal link, the summary agent first analyzes the influence path, including: (1) how changes in the knob value control the invocation of the target function and (2) the corresponding performance implications. Then, it formulates a tuning hypothesis based on the inferred causal relationships.

Example 4.3. Given the retrieved context of `innodb_spin_wait_delay`, the agent first generates an explanation of its influence path: "increasing this parameter makes `ut_delay()` inject longer delays per spin, reducing context switches but increasing CPU busy-wait; decreasing it shortens or skips `ut_delay()`, saving CPU but potentially increasing wakeups." The agent then formulates potential tuning strategies: "if runtime profiling shows `ut_delay()` consuming high CPU time, lowering `innodb_spin_wait_delay` is recommended to reduce spin overhead; if `sync_array_wait_event` dominates, increasing the delay helps avoid frequent context switch".

Finally, `SysInsight` constructs a tuning hypothesis set indexed by each knob k and its associated functions F^k, which records the promising knob adjustments given different function behaviors.

4.3 Tuning Rule Mining

This section presents how `SysInsight` generates quantitative tuning rules based on semantic tuning hypotheses. These rules are, in essence, inductive generalizations over a collection of successful tuning observations under the semantic guidance. The transformation involves two steps:

Semantic Sampling. `SysInsight` collects historical observations guided by semantic tuning hypotheses. It first identifies important knobs using diagnosis-based knob selection (Section 5.2.3), retrieves their associated hypotheses, and applies the recommended configurations. Each trial produces an observation that records: (i) the runtime context, (ii) the applied knob adjustment, and (iii) the observed performance impact. While a single successful observation could provide tuning guidance, its coverage is extremely limited because its contextual conditions are highly specific and rarely recur in future executions.

Inductive Rule Derivation. To obtain tuning guidance that generalizes across diverse contexts, `SysInsight` generates tuning rules (defined in Section 4.3.1) by mining the common patterns shared by successful observations. For example, if decreasing knob x by ~10 consistently improves performance when the sampling rate of function A appears at 2.5%, 3%, and 4%, `SysInsight` infers that the adjustment is beneficial within a broader interval such as $[2\%, 4\%]$. Using a customized Association Rule Mining (ARM) procedure (see Section 4.3.2), `SysInsight` consolidates the consistent contexts and adjustment magnitude into tuning rules that achieve both high coverage and high confidence across heterogeneous workloads.

4.3.1 Tuning Rule Definition.

Definition 1. *(Tuning Rules). A tuning rule $\mathcal{R}$ specifies how to adjust the knobs under certain system conditions. It can be expressed in the form $X \Rightarrow Y$, where X denotes an itemset of context predicates that should be satisfied, and Y denotes an itemset of recommended knob adjustments when X holds true. An itemset may consist of a single element or multiple conjunctive items. A tuning rule can be formally defined as:*

$$\left(\bigwedge_{k=1}^{K} p_k(w) \;\wedge\; \bigwedge_{j=1}^{N} r_j \in [l_j, u_j]\right) \Longrightarrow \left(\bigwedge_{i=1}^{M} \theta_i : [\mathit{adjustment}]\right) \quad (2)$$

$$\text{s.t.} \quad K + N \geq 0 \quad \text{and} \quad M > 0.$$

In this formulation, each $p_k(w)$ denotes a workload predicate, such as whether workload type = OLTP, the number of concurrent threads > 10, or query arrival rate exceeding a threshold. Each r_j denotes the proportion of time spent in function j, constrained within a range $[l_j, u_j]$. The symbols θ_i denote the tunable knobs, and $[adjustment]$ describes the recommended modification to each knob, such as decreasing `query_timeout` *by 10 seconds. This formulation indicates that when the context metrics meet the specified conditions, applying the recommended adjustments is expected to improve system performance.*

We adopt two metrics to evaluate the quality of tuning rules. To ensure that a rule is effective for future prediction, it must generalize across a sufficiently large number of situations (i.e., coverage) and demonstrate a high probability of making promising recommendations (i.e., confidence).

Definition 2. *(Coverage). Let $\mathcal{H} = \{h_1, \ldots, h_N\}$ denote a set of historical observations, and let $\mathcal{R}$ be a tuning rule expressed as $X \Rightarrow Y$. The **coverage** of $\mathcal{R}$ is defined as the proportion of observations in which the antecedent X holds:*

$$\mathit{Coverage}(\mathcal{R}) = \frac{|\{h \in \mathcal{H} \mid X \text{ holds in } h\}|}{|\mathcal{H}|}.$$

Definition 3. *(Confidence). Let $\mathcal{H}$ and $\mathcal{R}$ be as above. The **confidence** of $\mathcal{R}$ is defined as the proportion of observations in which the consequent adjustment Y improves the objective function f when antecedent X holds conditioned on all observations where the antecedent X holds and Y is applied:*

$$\mathit{Confidence}(\mathcal{R}) = \frac{\left|\left\{h \in \mathcal{H} \mid \left(X \text{ and } Y\right) \text{ and } \left(f \text{ is improved}\right) \text{ hold in } h\right\}\right|}{|\{h \in \mathcal{H} \mid X \text{ and } Y \text{ hold in } h\}|}.$$

4.3.2 Tuning Rule Induction. To induce tuning rules with high coverage and confidence, we formulate it as an association rule mining (ARM) problem [4], which aims to discover frequent co-occurrence patterns of contextual conditions and tuning actions that consistently lead to performance improvements. However, applying classical ARM techniques to database tuning is non-trivial due to two key challenges: *C2.1. Data sparsity and generalization:* Benchmarking databases is costly, resulting in few observations. Rules learned from limited data risk overfitting and are hard to generalize across hardware and workload scales. For instance, increasing `buffer_pool_size` by 10 GB may work on large-memory instances but fail on smaller ones. *C2.2. High-dimensional numerical data.* Knob values and runtime metrics are continuous, while traditional ARM handles categorical data. Naive discretization leads to a combinatorial explosion, high time complexity, and excessive memory usage [28]. To address these challenges, we propose a customized ARM pipeline for database tuning rule mining.

Step1: Pairwise Data Augmentation. To overcome the sparsity of raw observations and enhance rule robustness, we construct pairwise records (i.e., transactions in ARM) from historical observations. Given n historical observations, we enumerate all distinct pairs (h_i, h_j). If h_i outperforms h_j, we create an augmented record capturing: (1) the context predicates of h_j, (2) the knob adjustments in h_i that transition from the lower- to higher-performing configuration. To localize promising actions, only pairs with performance improvement exceeding a threshold are retained.

Step 2: Generalization-Oriented Encoding. We encode knob adjustments in a normalized and scale-invariant manner. Memory-related knobs are expressed as percentages of total memory to enhance comparability across different resource capacities. Knobs with wide numeric ranges are log-scaled, and those with smaller ranges are normalized using min-max scaling. Moreover, each adjustment is represented in both relative form (indicating the direction and magnitude of change) and absolute form (specifying the target value). These encodings prevent the loss of meaningful tuning actions that may manifest differently across various scales and contexts, improving the discovery of robust patterns.

Step 3: Target-Constrained FP-Growth. To address the challenges of high-dimensional numerical data, we introduce performance-aware discretization and tuning-specific constraints. Specifically, we employ the Least Squared Ordinate-Directed Impact Measure (LSQM) [29] to partition continuous features (e.g., knob adjustments and function sampling rates) into intervals that best capture their impact on performance. The discretization clusters continuous knob-adjustment magnitudes into performance-consistent intervals rather than preserving exact values. For example, adjustments of 1%, 12%, and 30% may fall into the same LSQM interval if they exhibit similar performance effects. Therefore, the mined rules represent ranges of effective adjustments rather than relying on repeated occurrences of the same numeric magnitude. For pattern mining, we adopt FP-Growth [21], which features a compact FP-Tree structure and eliminates the need for candidate generation, compared to Apriori-based methods [22]. We further enhance this process with *target itemset pruning*, which restricts the search to candidate branches containing at least one tuning-related knob adjustment. Algorithm 1 outlines the workflow of the enhanced FP-Growth procedure. It constructs a compressed prefix tree (FP-Tree) from the transaction database and recursively builds conditional FP-Trees to discover frequent itemsets. During the recursive exploration, branches that do not include any target items are pruned early (Lines 10–13), which significantly reduces the cost of constructing and traversing conditional FP-Trees.

Algorithm 1: FP-Growth with Target Item Set Pruning

Input: $Tree$: FP-Tree; $prefix$: current prefix ; $target$: set of target items; $min\_coverage$
Output: Frequent itemsets containing any item in $target$

```
1  if Tree is a single path then
2      foreach combination in all combinations of Tree do
3          candidate ← prefix ∪ combination;
4          if target ∩ candidate ≠ ∅ then
5              output candidate;
6  else
7      foreach item i in frequent items (ascending support) do
8          nprefix ← prefix ∪ {i};
9          P ← extract_conditional_pattern_base(Tree, i);
10         P' ← ∅;
11         foreach path p in P do
12             if (target ∩ nprefix ≠ ∅) or (target ∩ p ≠ ∅)
                then
13                 P' ← P' ∪ {p};
14         if P' not empty then
15             T' ← build_FP_Tree(P', min_coverage);
16             FP-Growth(T', nprefix, target, min_coverage);
```

5 CONFIGURATION RECOMMENDER

5.1 Diagnosis-Based Knob Selection

Unlike existing knob selection approaches that primarily rely on extensive configuration observations [6, 9, 17] or workload-specific heuristics [31] (discussed in Section 2.1), `SysInsight` adopts a *diagnosis-driven strategy* that enhances both the precision and interpretability of knob selection. The key idea is to first diagnose bottleneck functions that contribute to performance degradation and then determine which knobs to tune based on explicit function–knob mappings (Section 4.1). Adjustments to these mapped knobs can propagate through program logic, affecting the behavior of the identified bottleneck functions, which enables targeted interventions to restore performance.

To achieve fine-grained identification of function-level performance bottlenecks, `SysInsight` leverages Linux performance analysis tools (e.g., `perf` [3]) to sample stack traces. While more sophisticated and lightweight profiling techniques such as eBPF [14] have been proposed, they are orthogonal to our focus and not explored in this work. Instead of relying on manual inspection of visualizations such as flame graphs [20], `SysInsight` automates bottleneck diagnosis by combining two complementary methods:

Differential Profiling. When representative normal and degraded performance states are available, `SysInsight` compares the sampling rates of each function across these two conditions.

Functions that exhibit substantial variants in sampling proportion under degraded performance are flagged as likely contributors to performance issues (i.e., bottleneck functions).

SHAP-Based Profiling. In scenarios where a clear normal baseline is difficult to define, SysInsight leverages SHAP (SHapley Additive exPlanations) [35] to quantify the marginal contribution of each function to performance degradation based on historical execution data. SHAP is particularly well-suited for bottleneck identification as it provides a principled, game-theoretic allocation of the performance discrepancy across all functions while rigorously accounting for inter-feature dependencies. This global attribution framework enables the precise isolation of functions that consistently contribute to performance degradation, even under strong feature correlations. Specifically, SysInsight fits a regression model on the observed historical function sampling rates and the corresponding database performance. Given the current sampling rates, it computes SHAP values for each function and ranks them based on their negative contributions to overall database performance, which enables identification of bottleneck functions without a predefined performance baseline.

For each function f flagged as a potential bottleneck, SysInsight queries the function-knob mappings to identify its controlling knobs K^f as selected knobs. In addition, SysInsight matches the current runtime context against the antecedents of historical tuning rules; when a match is found, the knobs specified in the corresponding consequents are also incorporated into the set of selected knobs.

5.2 Rule-Augmented Tuning

Directly relying on an LLM to generate configuration recommendations often results in incorrect or suboptimal outcomes due to hallucinations. To mitigate this issue, SysInsight incorporates verified rules to enhance recommendation reliability.

5.2.1 Rule Retrieval and Maintenance. SysInsight retrieves relevant rules to guide the tuning process and continuously updates their statistics based on new observations to ensure reliability.

Rule Retrieval. To identify candidate rules, SysInsight first gathers all rules whose antecedent predicates match the current runtime context, referred to as *relevant rules*. Given the limited input window of the LLMs, SysInsight ranks these rules based on their *expected improvement* and retrieves only the top-k highest-scoring rules.

Definition 4. *(Expected Improvement). Let $\mathcal{S}(\mathcal{R})$ denote the set of observations where applying $\mathcal{R}$ led to performance improvement, i.e., $\mathcal{S}(\mathcal{R}) = \{h \in \mathcal{H} \mid (X \text{ and } Y) \text{ and } (f \text{ is improved}) \text{ hold in } h\}$. The **expected improvement** of rule $\mathcal{R}$ is defined as the product of the average performance gain it achieves and the probability (confidence) that it leads to performance improvement:*

$$EI(\mathcal{R}) = \frac{\sum_{h \in \mathcal{S}(\mathcal{R})} improvement(h)}{|\mathcal{S}(\mathcal{R})|} \times confidence(\mathcal{R}).$$

Rule Maintenance. After the suggested configuration is applied and its performance impact is observed, SysInsight updates the statistics of relevant rules. Among these relevant rules, SysInsight identifies those whose prescribed knob adjustments match the actual applied adjustments, referred to as *hit relevant rules*. If the applied adjustment results in a performance improvement, both the numerator and denominator of the confidence metric for each hit relevant rule are incremented by 1; otherwise, only the denominator is incremented. The *expected improvement* of each hit relevant rule is then recalculated to incorporate the new observation.

5.2.2 Rule-Augmented Prompt Generation. While the tuning rules offer reliable guidance, they may not match the current conditions (i.e., no applicable rules are retrieved), or potentially better adjustments may remain undiscovered by the rules. SysInsight therefore adopts a strategy that considers both *exploitation* and *exploration*: (i) exploiting verified rules where applicable to ensure reliability, and (ii) exploring recommendations guided by semantic hypotheses, which provide direction and help avoid overly aggressive or random exploration. The prompt mainly consists of three components: (1) Task instruction: A high-level description of the objective and expected behavior of the tuning agent; (2) Task Information: Contextual information required to ground the recommendations, including the hardware, workload characteristics, identified bottleneck functions, and the selected knobs; (3) Tuning Hypotheses and Rules: A set of semantic tuning hypotheses indexed by the selected knob and bottleneck functions, along with the retrieved rules.

5.2.3 Hypotheses-to-Rule Life-cycle. During the tuning process, SysInsight continuously transforms hypotheses into reliable rules. Given the hypotheses associated with the selected knobs and the retrieved tuning rules, SysInsight suggests promising configurations for evaluation. As it handles more tuning tasks, previously unseen knob-behavior patterns may emerge (e.g., new bottleneck-function interactions under varying workloads or hardware conditions). To capture these patterns, SysInsight periodically applies the customized ARM pipeline to induce new tuning rules from accumulated observations. Moreover, even when no applicable rules are available, SysInsight can still exploit hypotheses to discover superior configurations beyond the current rule set, which are subsequently validated and incorporated as new rules.

6 EVALUATIONS

6.1 Experiment Setting

Workload. Following the setup of GPTuner [31], we use TPC-H (OLAP type) with scaling factor 1 and TPC-C (OLTP type) with scaling factor 200. TPC-C uses ten terminals with unlimited arrival rate and the implementations of benchmarks are from BenchBase [13]. Additionally, we employ SYSBENCH [1] in read-write mode, configured with 100 tables each containing 6 million rows.

Hardware. We conduct experiments on two cloud instance configurations: (i) Instance A, with 4 CPU cores and 16 GB RAM, used for TPC-H and SYSBENCH workloads; and (ii) Instance B, with 16 CPU cores and 64 GB RAM, used for TPC-C workloads.

Baselines. We compare the performance of SysInsight with the state of the art tuning systems including manual-driven methods (GPTuner [31], DB-Bert [48]) and data-driven methods (SMAC [24], DDPG++[7], ResTune [54], OtterTune [6]).

Offline Data Collection. To provide historical data for both transfer-based data-driven methods (i.e., ResTune[54] and OtterTune [6]) and for SysInsight to mine tuning rules, 100 configurations are sampled per workload using Latin Hypercube Sampling [37]. To ensure a fair workload transfer scenario, all data collected under

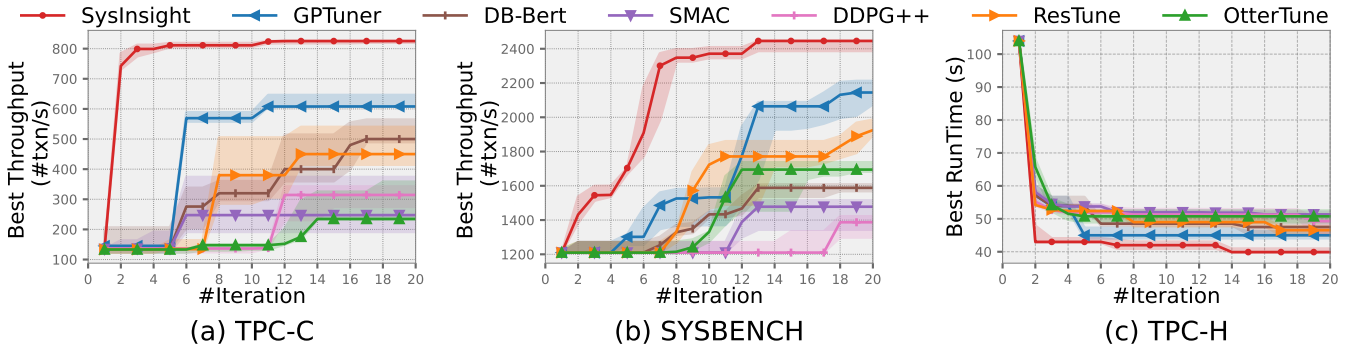


Figure 4: Best Performance Over Iterations. For TPC-C and SYSBENCH, top-left is better. For TPC-H, bottom-left is better.

the target workload is strictly excluded from the tuning process. For `SysInsight`, this means that the construction of tuning rules does not leverage any data from the target workload.

Tuning Setting. The target system is MySQL 8.0.36 (Community Edition, GPL license). Following prior studies [27, 31], all baselines tune 44 knobs with MySQL default configuration as the starting point and optimize throughput for OLTP workloads and total workload execution time for OLAP workloads. Note that production deployments often adopt non-default configurations, such as template settings (e.g., setting the buffer pool to 60% of available memory). Thus, the absolute gains compared with production-tuned settings are expected to be smaller. We run three tuning sessions for each method with 20 iterations per session. For `SysInsight` and GPTuner [31], we employ GPT-4o-mini as the backend model. As for DB-Bert, we adopt the authors' implementation which utilizes the pre-trained BERT model [12].

6.2 Efficiency Comparison

Figure 4 illustrates the performance of the best configurations found by different approaches throughout the tuning iteration. We report the median performance, with the interquartile range shown as shaded areas. `SysInsight` consistently identifies the best-performing configurations with significantly fewer iterations across all benchmarks. Compared with GPTuner, `SysInsight` converges to its best configuration on average 7.11× faster while still delivering an average performance improvement of 19.9%.

6.2.1 Comparison with Manual-driven Methods. GPTuner leverages LLMs to extract structured knowledge of each knob from documentation: (1) suggested range defined by *min_value* and *max_value*, (2) *suggested_values* and (3) *special_values*: Subsequently, it applies BO to explore a reduced and discretized search space defined by these ranges and values. Compared with data-driven methods, GPTuner effectively exploits prior knowledge from manuals to narrow the search space and accelerate convergence. Nevertheless, the structured knowledge extracted is limited and provides only moderate benefits. Among the 44 knobs, only 20 knobs have *min_value* or *max_value* differing from the MySQL defaults, 8 knobs include *suggested_values*, and 14 knobs have *special_values*.

The limitations mainly stem from two aspects. First, the knowledge provided by manuals is coarse-grained and lacks the precision guidance for fine-grained optimization. Second, for many knobs, universal information applicable to all scenarios simply does not exist, making it impossible to extract meaningful ranges or values without considering the context. According to the documentation of `innodb_spin_wait_delay`, "increasing this value can reduce CPU usage but may also increase response time, so it should be adjusted carefully based on system performance". While this qualitative insight is valuable, there is no well-defined range or suggested value that applies across all contexts, causing GPTuner to fail in extracting meaningful structured knowledge.

A similar limitation is observed in DB-Bert, which provides primarily context-agnostic guidance (with the exception of hardware-related factors) and derives meaningful knowledge for only 12 knobs. In contrast, `SysInsight` extracts hypotheses for all 44 knobs directly from the source code and has discovered tuning rules for 33 knobs with high confidence. This improvement arises because code-level analysis reveals fine-grained control information about each knob, providing richer and more context-aware references. Moreover, the tuning rules defined by `SysInsight` explicitly consider how each knob should be adjusted under different runtime contexts, enabling more targeted and effective optimization.

6.2.2 Comparison with Data-driven Methods. SMAC is shown to be the best-performing BO-based method for DBMS tuning without transferring according to the evaluation study [53]. DDPG++ is an improved version of the Deep Deterministic Policy Gradient (DDPG) algorithm, originally introduced in CDBTune [52] and enhanced in [7]. However, within 20 iterations, SMAC and DDPG++ fail to identify high-quality configurations because they start the search process from scratch. As for the baselines utilizing historical observations, OtterTune [6] identifies the most similar workload from a historical data repository through workload mapping and uses both matched data and target observations to train a surrogate model. ResTune [54] fits multiple base learners on observations from each source task and combines their predictions into a meta-learner via dynamic weighting. Compared with OtterTune, ResTune achieves better performance by mitigating negative transfer. However, both methods fundamentally rely on the similarity of performance surfaces across configurations, and their effectiveness degrades when the target workload is insufficiently similar to the source workloads. In contrast, `SysInsight` leverages insights from database internals—derived from source code and execution context—that capture generalizable relationships between system

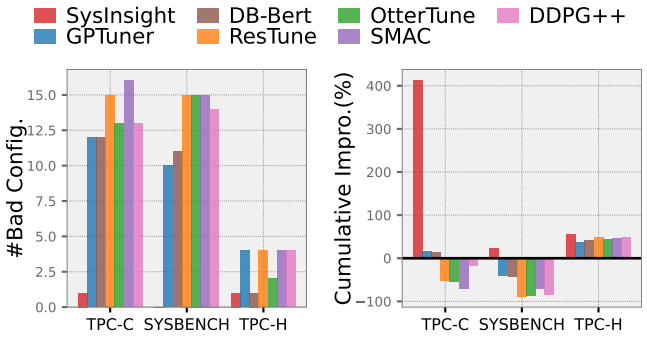
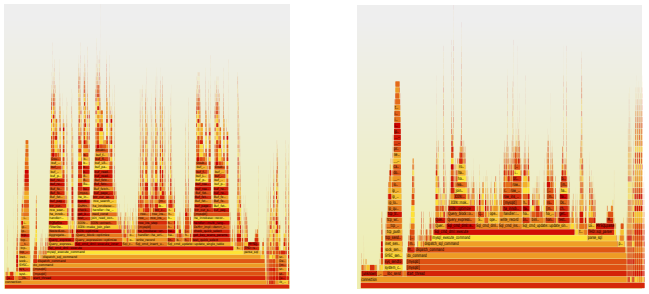


Figure 5: Reliability Comparison. Left figure shows #bad configurations (low is better). Right figure shows cumulative improvement over the default (higher is better).



(a) Default Configuration. **(b) Best Configuration.**

Figure 6: Different System Behaviors Shown in Flame Graphs. The width of each function represents its runtime cost.

behaviors and knob adjustments (e.g., observing frequent page misses suggests increasing the buffer pool), enabling robust and interpretable optimization beyond specific workloads.

6.3 Reliability Comparison

During tuning, it is crucial not only to eventually find the best-performing configuration but also to ensure that intermediate recommended configurations do not degrade system performance, referred as *reliability*. If the tuning algorithm explores too many poor configurations, the cumulative performance may fall below the default baseline, resulting in an overall performance loss. To quantify reliability, we adopt two complementary indicators: (1)#Bad Configurations: The number of recommended configurations that perform worse than the default. (2)Cumulative Improvement: The cumulative performance relative to the default during the tuning process. For OLTP workloads, it is measured by the total number of processed transactions (#txn), while for OLAP workloads, it is evaluated based on the total accumulated query execution time.

Figure 5 presents the two metrics for each method across the three workloads. `SysInsight` consistently achieves the fewest #Bad configurations, with only one instance reported in both TPC-C and SYSBENCH. For cumulative improvement, a positive value indicates that the tuning process yields a net performance gain over the default, while a negative value implies that the algorithm's exploration leads to overall performance degradation. SysInsight again demonstrates the best results, achieving a remarkable 412.8% improvement on TPC-C and maintaining positive gains on both SYSBENCH (24.2%) and TPC-H (19.8%). By contrast, other methods fail to amortize the cost of exploration on TPC-C and SYSBENCH.

6.4 Case Study

To further demonstrate the effectiveness of `SysInsight`, we conduct a case study using the TPC-C benchmark. We analyze the human effort required to formulate tuning hypothesis, illustrate how the identified bottlenecks guide knob selection, and how the retrieved tuning hypotheses and rules provide effective guidance.

Tuning Hypothesis Formulation. We analyze the human effort required for deriving the tuning hypothesis for `innodb_buffer_pool_size`. Static analysis identifies 42 functions that are control-dependent on this knob. A database expert spent approximately one full day tracing the knob's propagation and understanding its impact paths. For each of these 42 functions, it was typically necessary to examine an additional 10 surrounding or dependent functions to fully understand their behavior. This further illustrates that even analyzing a single knob involves considerable effort, and extending such analysis to dozens of knobs would be highly labor-intensive.

Diagnosis-based Knob Selection. As shown in Figure 6, under the default configuration, wide I/O hotspots dominate the runtime. `SysInsight` identifies functions such as `buf_page_get_gen` and `buf_LRU_get_free_block` as bottlenecks, indicating that the I/O path (disk page loading and LRU page allocation) is the key performance constraint. Additionally, other critical bottleneck functions such as `srv_purge_coordinator_thread` (responsible for undo log cleanup) and `sync_array_wait_event` (related to lock contention and spin-wait synchronization) are also identified. By querying the function–knob mapping, `SysInsight` selects `innodb_buffer_pool_size`, `innodb_purge_batch_size` and `innodb_spin_wait_delay` that these bottleneck functions are control-dependent on.

Rule-Guided Configuration Tuning. Based on the bottleneck diagnosis, `SysInsight` retrieves the corresponding hypotheses and tuning rules for the selected knobs from the knowledge library, as shown in Table 1. Those rules provide interpretable and context-aware guidance by linking observed bottlenecks to promising configuration adjustments, each accompanied by a confidence score that indicates the reliability. For example, while manual guidance suggests that setting `innodb_buffer_pool_size` to 70–80% of system memory yields good performance (as also adopted by GPTuner), `SysInsight` leverages validated high-confidence rules to dynamically adjust this knob based on runtime function-level signals (e.g., `buf_LRU_get_free_block`). As a result, `SysInsight` recommends a buffer pool size of around 50% of available memory, which achieves superior performance by reducing I/O overhead while preserving sufficient memory for other critical components.

6.5 Scalability Study

We study the impact of database size on tuning performance by conducting experiments on TPC-H workloads with scale factors of 1, 10, and 50. We compare `SysInsight` with three representative baselines from different categories: (i) GPTuner (manual-driven),

Table 1: Examples of Tuning Hypotheses and Rules. Each column presents a representative case where SysInsight selects tuning knobs based on bottleneck functions, retrieves tuning hypothesis and rules of high confidence score.

Function	`buf_LRU_get_free_block`	`srv_purge_coordinator_thread`	`ut_delay`, `sync_array_wait_event`
Associate Knob	`innodb_buffer_pool_size`	`innodb_purge_batch_size`	`innodb_spin_wait_delay`
Causal Link	`innodb_buffer_pool_size` controls memory space for pages in `buf_LRU_get_free_block()`. A small pool triggers frequent evictions and dirty page flushes.	`innodb_purge_batch_size` controls the purge batch size in `srv_purge_coordinator_thread()`. A small value slows undo log cleanup and lengthens MVCC chains; a large value can cause I/O spikes.	`innodb_spin_wait_delay` controls the spin duration in `ut_delay()`. A short delay leads to frequent context switches (`sync_array_wait_event()`); a long delay wastes CPU in spin-lock polling.
Hypothesis	If `buf_LRU_get_free_block()` call is frequent, increase `innodb_buffer_pool_size`.	If `srv_purge_coordinator_thread()` call is frequent and undo log buildup slows snapshot reads (`row_search_mvcc()`), increase the knob.	If `sync_array_wait_event()` call is frequent, increase the knob; if `ut_delay()` call is frequent but lock contention remains, decrease it.
Tuning Rule	r(`buf_LRU_get_free_block`) > 16% $\Rightarrow$ increase `innodb_buffer_pool_size` by (0.16Mem, 0.50Mem]	r(`srv_purge_coordinator_thread`) > 5% $\wedge$ r(`row_search_mvcc`) > 10% $\Rightarrow$ increase `innodb_purge_batch_size` by (0,100]	r(`sync_array_wait_event`) > 3% $\Rightarrow$ increase `innodb_spin_wait_delay` by (0,10]
Confidence	0.87	0.85	0.74

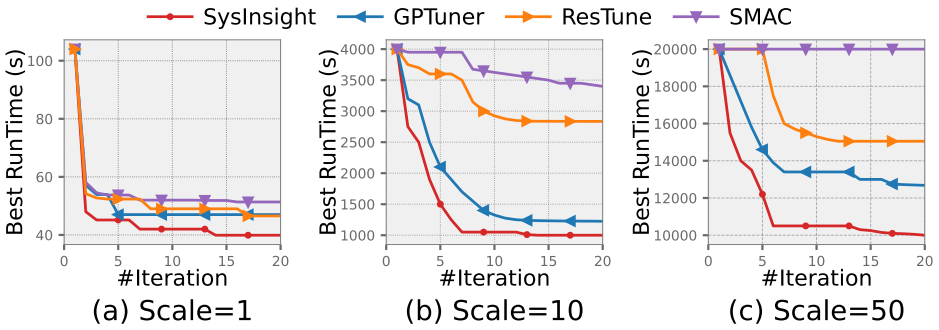


Figure 7: Effect of Database Size on Tuning Performance (bottom-left is better).

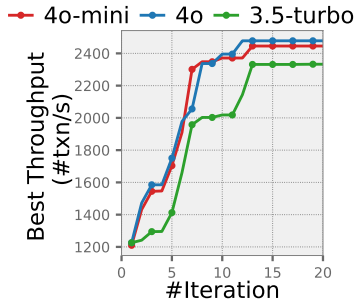


Figure 8: Effect of Language Models (top-left is better).

(ii) ResTune (data-driven with transfer learning), and (iii) SMAC (data-driven without transfer learning). Figure 7 shows the best runtime achieved versus the number of tuning iterations under different scales. As the database size grows, the absolute runtime of all methods increases, but `SysInsight` consistently achieves the fastest convergence and the lowest final runtime. The performance of data-driven methods degrades noticeably as the database grows, and SMAC fails to find any configuration better than the default at a scale factor of 50. This degradation occurs because larger datasets result in a sparser promising performance landscape, making it harder for data-driven methods to identify good configurations.

Unlike these baselines which rely solely on a single statistical metric (e.g., total execution time for TPC-H) to explore the configuration space, `SysInsight` leverages code-level diagnostics and causal hypotheses to conduct targeted tuning, thereby mitigating the challenges of sparse performance landscapes as database size grows. For example, `SysInsight` observes a bottleneck that `TableScanIterator::Read` incurs a high cost, while `buf_read_ahead_linear` is rarely invoked, indicating that the linear read-ahead mechanism is underutilized. From the function-knob mapping, `innodb_read_ahead_threshold` is identified as the controlling knob, and `SysInsight` lowers this threshold to activate read-ahead logic more aggressively, improving sequential scan performance. Similarly, `SysInsight` detects the high cost of hash join operations (e.g., `HashJoinIterator::BuildHashTable`) and doubles its associated knob `join_buffer_size` from 256KB to 512KB, which allows more intermediate join results to be retained in memory and accelerating hash table construction. As a result, `SysInsight` converges to an optimal configuration in fewer iterations, achieving a 47.5% runtime reduction at Scale = 50 within 5 iterations, consistently outperforming all baselines.

6.6 Analysis of SysInsight

6.6.1 Overhead Analysis. `SysInsight` incurs offline overhead from tuning hypothesis formulation and rule mining. Hypotheses are generated once per DBMS version, requiring 45 hours for function–knob extraction and 4 hours ($5) for LLM-based generation

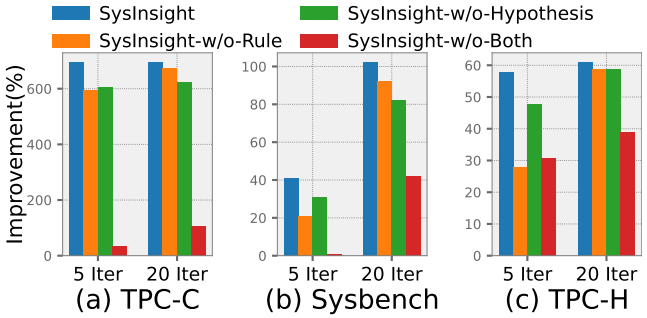


Figure 9: Ablation Study on Tuning Hypotheses and Rules.

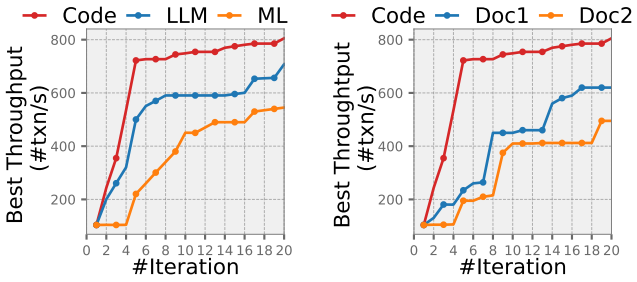


Figure 10: Knob Selection: We compare code-based, LLM-based and ML-based Methods.

Figure 11: Knowledge Source: We compare code-driven and manual-driven methods.

using 1.5M input and 121K output tokens. Rule mining involves collecting 300 configuration samples (20 hours) and executing the ARM pipeline (20 hours). All results are cached for reuse across future tuning tasks. The online tuning overhead consists of: (1) the time taken by the optimizer to suggest the next configuration, and (2) the workload evaluation time. On average, `SysInsight` spends about 26 seconds to produce the next configuration, including 0.03 seconds for diagnosis-based knob selection, 0.3 seconds for hypothesis and rule retrieval, and 25 seconds for the tuning agent's recommendation. In practice, workload evaluation time typically dominates the total tuning cost. As a result, recent works [27, 31] focus on *sample efficiency*, i.e., the number of DBMS evaluations required to achieve a certain level of performance improvement, rather than the absolute optimization time. In this regard, `SysInsight` demonstrates a significant advantage, as illustrated in Figure 4.

6.6.2 Effect of Different Language Models. As shown in Figure 8, we compared the performance of different LLMs, including gpt-4o, gpt-4o-min, and gpt-3.5-turbo, which are used as the backend model for tuning agents. As the capability of the underlying model improves, `SysInsight` achieves better results, indicating that `SysInsight` can effectively leverage the power of LLMs. Among them, gpt-4o-min performs comparably to gpt-4o while incurring only about 6% of the cost per token. Therefore, we adopt gpt-4o-min as the default backend model in `SysInsight` to achieve a favorable balance between performance and cost.

6.6.3 Effect of Hypotheses and Rules. Figure 9 shows the performance improvement against the default configuration of `SysInsight` when removing hypothesis retrieval, rule retrieval, or both, measured after 5 and 20 iterations. We observe that removing both components leads to a significant performance drop, while removing only one has a relatively smaller impact, indicating that the two components are complementary. Moreover, `SysInsight-w/o-hypothesis` performs better in early stages (5 iterations), suggesting that rules can quickly provide good configurations. In contrast, `SysInsight-w/o-rule` achieves better performance in later stages (20 iterations), indicating that hypotheses help explore potentially better configurations.

6.6.4 Comparison of Knob Selection Methods. We compare different knob selection methods by replacing the knob selection component of `SysInsight`. To isolate the effect of knob selection, we remove the rule retrieval module of `SysInsight`, which otherwise implicitly influences knob choices. The compared methods are: (1) Code-based: Our methods that identifies bottleneck functions and leverages static code analysis to find the associated knobs for tuning; (2) LLM-based: the approach used by GPTuner [31], prompts LLMs to select important knobs from the candidate set based on workload information; (3) ML-based: the approach used by Tuneful [17], which begins with tuning all knobs and removes 40% of the less important knobs every 5 iterations using Gini importance ranking derived from prior observations. Figure 10 shows the results on the TPC-C workload. The code-based method achieves the best performance due to its accurate identification bottleneck functions and associate knobs through code analysis. In contrast, the ML-based method performs worst, as the limited number of observations is insufficient to yield reliable knob importance rankings.

6.6.5 Comparison of Different Knowledge Sources. We investigate the impact of different knowledge sources on `SysInsight` by replacing its code-derived tuning hypotheses with alternative sources: structural knowledge from GPTuner [31] (denoted by Doc1) and tuning hints from DB-Bert [48] (denoted by Doc2). To avoid interference, the rule retrieval module of `SysInsight` is disabled during this comparison. As shown in Figure 11 on TPC-C, our code-based source achieves the best performance, providing context-aware and fine-grained knowledge compared with the structural knowledge or tuning hints extracted from database manuals.

7 CONCLUSION AND OUTLOOK

We presented `SysInsight`, a code-driven tuning system that derives fine-grained, context-aware, and verifiable tuning insights directly from DBMS source code. Unlike prior works, `SysInsight` takes a concrete step toward white-box tuning by explicitly leveraging system internals to guide the optimization process. Extensive experiments demonstrate its efficiency, reliability and scalability. While `SysInsight` focuses on code-driven tuning, we acknowledge that other sources—such as text documentation and historical observations—can be complementary. As future work, we plan to explore hybrid strategies that unify these sources to further enhance tuning agents' adaptability and effectiveness.

REFERENCES

[1] 2015. Sysbench. https://github.com/akopytov/sysbench.

[2] 2019. MySQL Tuning Primer Script. https://github.com/major/MySQLTuner-perl.

[3] 2024. *perf: Linux profiling with performance counters.* https://perfwiki.github.io/main/.

[4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 207–216.

[5] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. Morgan Kaufmann, 1110–1121.

[6] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD Conference*. ACM, 1009–1024.

[7] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253.

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *PACT*. ACM, 303–316.

[9] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *SIGMOD Conference*. ACM, 646–659.

[10] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions. *Proc. VLDB Endow.* 14, 8 (2021), 1401–1413.

[11] Surajit Chaudhuri and Gerhard Weikum. 2006. Foundations of Automated Database Tuning. In *VLDB*. ACM, 1265.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.

[13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[14] Meizhu Du and Xinfeng Shu. 2023. Automatic eBPF program generation method for performance monitoring. In *AIPR*. ACM, 1255–1261.

[15] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[16] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.

[17] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune?: In Search of Optimal Configurations for Data Analytics. In *KDD*. ACM, 2494–2504.

[18] Ayat Fekry, Lucian Carata, Thomas F. J.-M. Pasquier, Andrew Rice, and Andy Hopper. 2020. Tuneful: An Online Significance-Aware Configuration Tuner for Big Data Analytics. *CoRR* abs/2001.08002 (2020).

[19] Jia-Ke Ge, Yanfeng Chai, and Yunpeng Chai. 2021. WATuning: A Workload-Aware Tuning System with Attention-Based Deep Reinforcement Learning. *J. Comput. Sci. Technol.* 36, 4 (2021), 741–761.

[20] Brendan Gregg. 2024. *Flame Graphs.* https://www.brendangregg.com/flamegraphs.html.

[21] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. 2004. Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. *Data Min. Knowl. Discov.* 8, 1 (2004), 53–87.

[22] Jeff Heaton. 2017. Comparing Dataset Characteristics that Favor the Apriori, Eclat or FP-Growth Frequent Itemset Mining Algorithms. *CoRR* abs/1701.09042 (2017).

[23] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2 (2025), 42:1–42:55.

[24] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (Lecture Notes in Computer Science)*, Vol. 6683. Springer, 507–523.

[25] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024).

[26] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *HotStorage*. USENIX Association.

[27] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proc. VLDB Endow.* 15, 11 (2022), 2953–2965.

[28] Minakshi Kaushik, Rahul Sharma, Iztok Fister Jr., and Dirk Draheim. 2023. Numerical Association Rule Mining: A Systematic Literature Review. *CoRR* abs/2307.00662 (2023).

[29] Minakshi Kaushik, Rahul Sharma, Sijo Arakkal Peious, and Dirk Draheim. 2021. Impact-Driven Discretization of Numerical Factors: Case of Two- and Three-Partitioning. In *BDA (Lecture Notes in Computer Science)*, Vol. 13147. Springer, 244–260.

[30] Jan Kossmann and Rainer Schlosser. 2020. Self-driving database systems: a conceptual approach. *Distributed Parallel Databases* 38, 4 (2020), 795–817.

[31] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proc. VLDB Endow.* 17, 8 (2024), 1939–1952.

[32] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*. IEEE Computer Society, 75–88.

[33] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130.

[34] Yiyan Li, Haoyang Li, Pu Zhao, Jing Zhang, Xinyi Zhang, Tao Ji, Luming Sun, Cuiping Li, and Hong Chen. 2024. Is Large Language Model Good at Database Knob Tuning? A Comprehensive Experimental Evaluation. *CoRR* abs/2408.02213 (2024).

[35] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.

[36] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD Conference*. ACM, 631–645.

[37] Michael D. McKay. 1992. Latin Hypercube Sampling as a Tool in Uncertainty Analysis of Computer Models. In *WSC*. ACM Press, 557–564.

[38] Morgane Menz, Sylvain Dubreuil, Jérôme Morio, Christian Gogu, Nathalie Bartoli, and Marie Chiron. 2020. Variance based sensitivity analysis for Monte Carlo and importance sampling reliability assessment with Gaussian processes. *CoRR* abs/2011.15001 (2020).

[39] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *ICSE*. ACM, 97:1–97:13.

[40] Stefano Nembrini, Inke R. König, and Marvin N. Wright. 2018. The revival of the Gini importance? *Bioinform.* 34, 21 (2018), 3711–3718.

[41] PG-Tune [n.d.]. PGTune. https://pgtune.leopard.in.ua.

[42] Dennis E. Shasha and Philippe Bonnet. 2002. Database Tuning: Principles, Experiments, and Troubleshooting Techniques. In *VLDB*. Morgan Kaufmann.

[43] Dennis E. Shasha and Steve Rozen. 1992. Database Tuning. In *VLDB*. Morgan Kaufmann, 313.

[44] Adam J. Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *VLDB*. ACM, 1081–1092.

[45] David G Sullivan, Margo I Seltzer, and Avi Pfeffer. 2004. Using probabilistic reasoning to automate software tuning. *ACM SIGMETRICS Performance Evaluation Review* 32, 1 (2004), 404–405.

[46] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source Code Summarization in the Era of Large Language Models. In *ICSE*. IEEE, 1882–1894.

[47] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[48] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 190–203. https://doi.org/10.1145/3514221.3517843

[49] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proceedings of the International Conference on Very Large Data Base* 14, 13 (2021), 3402–3414.

[50] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. 2023. ConfTainter: Static Taint Analysis For Configuration Options. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1640–1651.

[51] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. 2002. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB*. Morgan Kaufmann, 20–31.

[52] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD Conference*. ACM, 415–432.

[53] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (2022), 1808–1821.

[54] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD Conference*. ACM, 2102–2114.

[55] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. *CoRR* abs/2203.14473 (2022).

[56] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 539–552. https://doi.org/10.14778/3632093.3632114

[57] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *ISSTA*. ACM, 1592–1604.

[58] Shulin Zhou, Xiaodong Liu, Shanshan Li, Wei Dong, Xiangke Liao, and Yun Xiong. 2016. ConfMapper: Automated Variable Finding for Configuration Items in Source Code. In *QRS Companion*. IEEE, 228–235.

[59] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 symposium on cloud computing*. 338–350.

Supplemental Material

OUTLINE

This supplemental material is organized as follows.
S1. Statistics and More Evaluation on Tuning Rules.
S2. Evaluation on PostgreSQL.
S3. Human Effort Analysis for Tuning Hypothesis Formulation.
S4. Discussion on Future Extensions.

S1 TUNING RULES

S1.1 Tuning Rule Statistics

Figure S1 plots the coverage and certainty over (i) all extracted rules, and (ii) the per-knob distributions for the top-5 most frequently occurring knobs in the rule set. The coverage of a rule reflects the fraction of samples in the trace repository that satisfy its context predicates, and may therefore not be very high because these predicates cannot be too broad without losing their discriminative power. In contrast, the certainty measures the fraction of the matched samples that observe performance improvement when the rule is applied, which tends to be higher for reliable rules. We note that our ARM procedure enforces minimum thresholds of coverage ≥ 0.20 and certainty ≥ 0.50 during rule mining, which define the lower bounds observed in the figure. Despite the certainty threshold, the mined rule set contains many high-quality rules with substantially higher certainty.

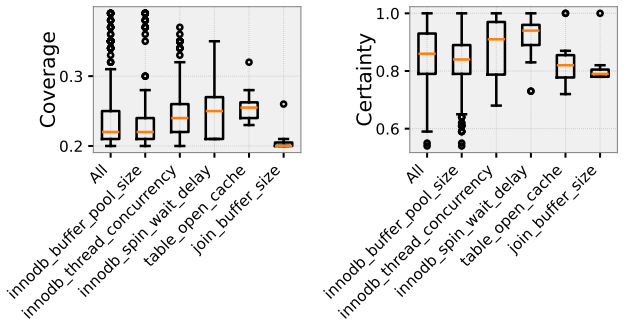


Figure S1: The Distribution of Coverage and Certainty across All Mined Rules, along with Per-Knob Distributions for the Top-5 Most Frequent Knobs.

S1.2 Ablation on Rule Scale

To analyze the effect of tuning rules on SysInsight, we conduct an ablation on the rule scale by limiting the number of rules to 0%, 10%, 50%, and 100% of the full rule set. Figure S2 reports the tuning improvement against the default setting at iterations 5, 10 and 20 on Sysbench and TPC-H workloads. When only 50% of the rules are retained, SysInsight can already achieve performance close to the full-rule setting. This indicates the presence of redundancy, where multiple similar rules may apply to a given situation. For example, consider the following three rules:

Rule 1: buf_LRU_get_free_block > 16% → increase buffer_pool_size by 10% Mem

Rule 2: buf_LRU_get_free_block > 32% → increase buffer_pool_size by 10% Mem

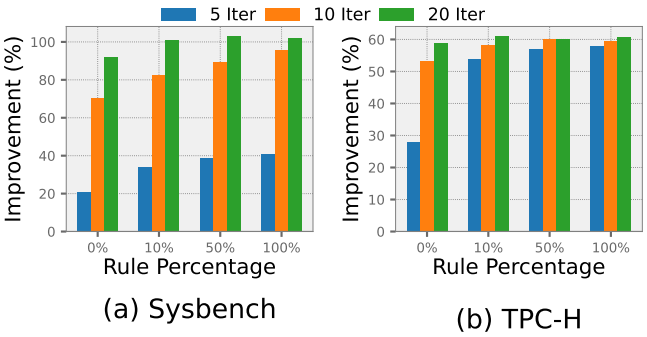


Figure S2: Effect of Rule Scale.

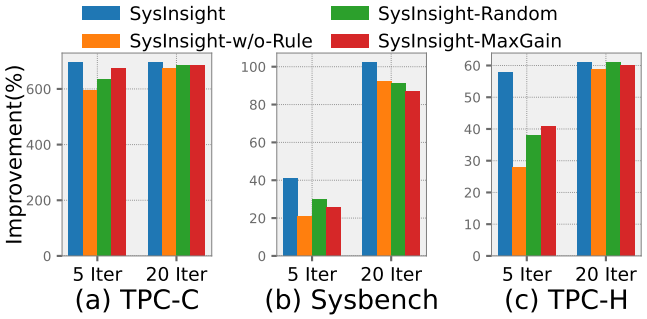


Figure S3: Evaluation on Reliability-Aware Rule Retrieval.

Rule 3: buf_LRU_get_free_block > 32% & Buf_fetch > 4% → increase buffer_pool_size by 10% Mem

For a tuning context (e.g., buf_LRU_get_free_block=33% and Buf_fetch=6%), the three rules are all applicable. Such redundancy naturally arises from the rule mining process: the ARM pipeline extracts all rules whose coverage and certainty exceed the preset thresholds. To manage this redundancy, SysInsight employs a rule retrieval mechanism that prioritizes applicable rules with higher expected improvement, derived from both their certainty and the historical average performance gain.

Overall, the scale ablation demonstrates that a relatively small portion of rules can already yield competitive tuning performance. Nevertheless, maintaining a larger rule pool improves robustness and generalization across diverse tuning contexts.

S1.3 Evaluation on Rule Retrieval

We further isolate and evaluate the reliability scoring for rule retrieval. We compare four variants: (1) SysInsight: retrieves the matched rules ranked by their expected improvement, which is computed from rule certainty and the average historical performance gain; (2) SysInsight-Random: randomly selects the matched rules from the candidate set; (3) SysInsight-MaxGain: retrieves the matched rules ranked by their average historical gain, ignoring certainty; (4) SysInsight-w/o-Rule: disables the rule retrieval mechanism entirely and relies solely on the learned tuning hypotheses. Figure S3 shows the result. Both SysInsight-Random and

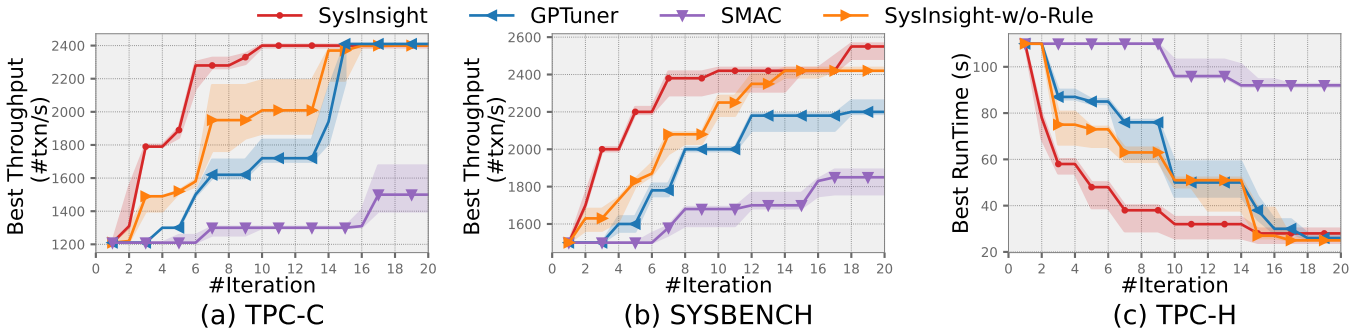


Figure S4: Best Performance Over Iterations on PostgreSQL.

SysInsight-MaxGain perform worse than the full SysInsight, demonstrating that the reliability scoring is essential for prioritizing high-quality rules and achieving better tuning results.

S2 EVALUATION ON POSTGRESQL

To demonstrate the generality of SysInsight, we have ported SysInsight to PostgreSQL and verified its applicability end-to-end. Specifically, PostgreSQL exposes its configuration variables as global C variables in guc.c. We compile PostgreSQL into LLVM bitcode and apply the same LLVM IR–based function–knob mapping procedure as in MySQL: (1) compile PostgreSQL into .bc, (2) mark GUC variables as taint sources, and (3) perform data-/control-flow taint propagation to obtain a function-level dependency graph for each knob. Aggregating affected functions yields the complete PostgreSQL function–knob mapping. The subsequent stages of SysInsight (code-driven hypothesis extraction and rule mining) remain effective with only minimal adaptation.

We evaluate SysInsight on PostgreSQL under the same workload setting and tune 48 knobs, comparing its performance against GPTuner, SMAC, and SysInsight-w/o-Rule. Figure S4 shows a consistent performance trend as observed on MySQL: SysInsight achieves the best or similar final performance and converges significantly faster. While SysInsight-w/o-Rule requires more exploration time due to the lack of rule-based priors, it still outperforms GPTuner on TPC-C and Sysbench, and achieves comparable performance on TPC-H. This performance gain stems from two main factors: First, it identifies and tunes knobs that control runtime bottleneck functions, where the bottlenecks are detected online and the corresponding knobs are mapped via static code analysis. Second, the code-derived hypotheses provide context-aware directional guidance during tuning, enabling the system to make informed decisions based on the runtime context. In contrast, GPTuner relies on manually written database documentation to construct its tuning space, which remains insufficiently informed. Among the 48 knobs, only 13 have suggested ranges that differ from the PostgreSQL defaults, and only 12 provide suggested values derived from PostgreSQL's own default configuration. Instead, SysInsight extracts a substantially richer set of tuning hypotheses from the DBMS source code, offering broader and deeper guidance across diverse workload conditions.

S3 HUMAN EFFORT ANALYSIS FOR TUNING HYPOTHESIS FORMULATION

We conduct an analysis estimating the human effort required to manually extract the tuning knowledge for all the 44 target MySQL knobs, as a comparison to the LLM agent in SysInsight. Given a function affected by a knob, the LLM agent first inspects its parent and surrounding context and is prompted to assess whether the available code is sufficient to understand the knob's behavior. If not, it automatically invokes search_function(f) or search_class(cls) to resolve unclear API calls and dependencies, recursively retrieving additional code context. Across all the target 44 MySQL knobs, this process leads to the retrieval and analysis of approximately 200,000 lines of source code. Manually performing this level of cross-file, multi-hop reasoning would require an estimated 35 full workdays for a DBMS expert, assuming a conservative reading speed of five seconds per line for the low-level storage engine code. In contrast, the LLM completes the same analysis in approximately four hours (at a cost of around five US dollars), while producing consistent and structured tuning hypotheses.

S4 DISCUSSION ON FUTURE EXTENSIONS

While SysInsight focuses on efficient online configuration tuning, its design naturally opens the door to two promising extensions: (a) incremental data collection to automatically strengthen tuning knowledge, and (b) automated generation of concise manuals or runbooks to support human-in-the-loop tuning. We briefly discuss these opportunities below.

Incremental Data Collection. SysInsight already exhibits a preliminary form of targeted data collection during the rule derivation phase: it samples runtime observations conditioned on the tuning hypotheses. This sampling is guided by semantically meaningful knob-function hypotheses extracted from code, which enables the system to focus on regions of the configuration space that are likely to yield useful insights. This mechanism could be extended into a more explicit active data collection framework, where the system detects low-coverage or low-certainty hypotheses and incrementally collects more data to improve them, which is a promising direction for future work.

Auto-Generated Manuals. The verified rules in SysInsight are already expressed in a declarative, human-readable form (e.g., "when function A's sampling rate is below 3%, increase knob X by 10%"). Currently, the tuning agent retrieves the rule that matches

the current with highest expected improvement since the total number of mined rules can be large. For example, the knob innodb_buffer_pool_size is found to explicitly control 42 functions, and our association rule mining (ARM) process further uncovers additional implicit relationships. Because the space of possible contextual predicates (e.g., functions and corresponding metrics) is large, the ARM procedure generates many fine-grained rules with different coverage and confidence. To produce concise and human-readable manuals, rule consolidation and summarization will be necessary. However, this process involves a trade-off, as consolidation may obscure the fine-grained context-specific insights encoded in the original rules. We therefore regard automated manual/runbook construction as another compelling direction for future work.