ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/adithya-sirimalla/642

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 1 & Iss: 4

Research Article

Autonomous Performance Tuning Framework for Databases Using Python and Machine Learning

Adithya Sirimalla*

Citation: Sirimalla A. Autonomous Performance Tuning Framework for Databases Using Python and Machine Learning. *J Artif Intell Mach Learn & Data Sci 2023* 1(4), 3139-3147. DOI: doi.org/10.51219/JAIMLD/adithya-sirimalla/642

Received: 03 October, 2023; **Accepted:** 18 October, 2023; **Published:** 20 October, 2023

***Corresponding author:** Adithya Sirimalla, Enliven Technologies Inc., USA, E-mail: adithya.sirimalla@gmail.com

ABSTRACT

The increasing complexity and dynamic workload of contemporary database systems require strong and dynamic performance-tuning mechanisms. Manual tuning methods are time-consuming, prone to human error, and cannot keep up with the changing needs of the system. This study suggests an independent system for the performance optimization of databases based on the strengths of Python and sophisticated machine learning methods. In our structure, real-time database monitoring, smart data analysis, and ML-based decision-making are combined and used to automatically detect performance bottlenecks and implement the most effective configuration changes. In particular, it uses reinforcement learning or other appropriate ML algorithms to learn system and workload behaviors and predict and act upon tuning behaviors, such as index management, parameter settings, and resource allocation. The structure is formed in a way that it will constantly learn and keep changing, which will guarantee high performance in diverse operating conditions. We elaborate on the architectural elements, ML applications, and Python implementation plan, which will essentially lessen operational overheads, improve database response, and boost the overall responsiveness of the system. The effectiveness of the framework was experimentally evaluated and shown to be effective in realizing significant performance improvements over traditional tuning techniques.

Keywords: Autonomous Database Tuning, Performance Optimization, Machine Learning, Reinforcement Learning, Database Management Systems

1. Introduction

1.1. Background and motivation

Database performance has become a crucial backbone of contemporary applications, especially when organizations rely on OLTP and OLAP workloads of large volumes. The results directly affect the query latency, throughput, and operational cost through efficient tuning[1]. Nonetheless, tuning has become difficult as DBMSs have hundreds of interacting parameters that are nonlinear when tested with dynamic loads[2].

Conventional tuning is highly dependent on human knowledge, trial and error, and heuristics. This method is slow, prone to errors, and cannot match fast-changing cloud-native and distributed data ecosystems[3]. Even highly skilled DBAs cannot ensure consistent performance in environments with heterogeneous architectures because of changes in workloads that become unpredictable[4].

Machine learning has changed the way performance optimization is performed. Similar systems, such as OtterTune,

have shown that it is possible to use learned models to solve the problem of tuning decisions better than expert tuning using workload traces[5]. Based on reinforcement learning, methods such as CDBTune+[6] and adaptive multi-model RL agents[7] have extended the limits of the performance of systems that investigate and train tuning techniques autonomously.

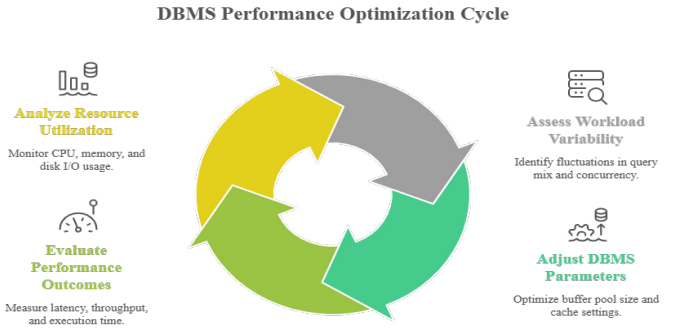

Figure 1: Provides a conceptual overview of how workload dynamics, system parameters, and performance outcomes interact, highlighting the inherent complexity of the manual performance tuning.

Despite these advances, gaps remain. ML-driven tuners that can be used in production are still constrained by a lack of transferability, heavy dependency on system-specific hooks, or failure to respond dynamically to changing workloads[8].

To establish the context of these gaps, **(Table 1)** compares manual tuning and tuning based on MLs and presents the trade-offs applicable in real-world database settings.

Table 1: Comparison of Manual vs. ML-Based Database Tuning Approaches.

Aspect	Manual Tuning	ML-Based Tuning
Adaptability to Changing Workloads	Low	High
Dependence on Expertise	High	Moderate to Low
Scalability	Limited	Strong
Error Probability	High	Low (with correct training)
Speed of Optimization	Slow	Fast / Automated
Cross-DBMS Generalization	Weak	Potentially Strong

Such a comparison highlights the necessity of a general-purpose, Python-based autonomous system that can monitor workloads, learn tuning policies, and safely apply performance enhancements to a wide range of DBMS settings.

1.2. Problem statement

Most current ML-based tuning systems have one or more severe constraints.

- Relying on proprietary internal components of the DBMS, which restrict portability[9].
- Low flexibility in shifting dynamic workloads[10].
- Complex training requirements arise from the use of large amounts of historical data[8].
- The absence of safe tuning mechanisms exposes systems to possible performance regressions[11].

In summary, there is no modular, cross-platform, Python-centric autonomous tuning system that can continuously monitor system behavior, learn the best system settings, and make tuning decisions safely and automatically in a manner that does not require human intervention.

1.3. Research objectives

This study aims to create and test a fully autonomous performance tuning system using Python and machine learning. The objectives are as follows:

- Development of a real-time performance monitoring modular architecture.
- Creating a hybrid ML pipeline for workload-aware tuning: supervised learning in combination with reinforcement learning.
- Application of a safe rollback-capable tuning action executor.
- Assuring cross-DBMS compatibility using Python-based connectors to the database.
- The performance improvement with various workloads was tested in a real situation.

1.4. Contributions

The main contributions of this study are as follows:

- A DBMS-independent self-tuning system that can be applied universally to PostgreSQL, MySQL, and NoSQL databases.
- An intelligent learning system that combines performance prediction models with adaptation using reinforcement learning[12,4].
- A tuning floor module that provides rollback logic based on safe RL principles[11].
- An overall assessment of the proposed structure with default settings, hand-tuning, and state-of-the-art ML-based tuners.

All these contributions address the fundamental drawbacks of the current systems and contribute to the state-of-the-art research in the field of self-driving databases.

1.5. Paper organization

The remainder of this paper is organized as follows.

- **Part 2:** In this section, classic tuning methods, machine learning-based optimization methods, and autonomous database management systems are reviewed.
- Section 3 discusses the proposed autonomous performance-tuning framework using Python.
- Section 4 provides the experimental configuration and description of the workloads and analysis.
- Section 5 presents the findings, discussing the results of the performance improvement and model behavior.
- Section 6 summarizes the study and identifies future research directions.

2. Related Work

2.1. Conventional database tuning algorithms

Traditionally, the tools used to tune the performance of databases have been based on manual administration, expert heuristics, and rule-based systems. The initial relational DBMSs had fixed configuration parameters that were manipulated by trial and error by an administrator using performance reports.

It involved an in-depth understanding of the system and had a limitation of excessive dimensionality of the current database settings[13]. The rule-based optimizers attempted to turn this process into a standard process using cost models to make indexing, join order, and access path decisions. These systems did not work well, even though they were designed with structured guidance; however, heuristics could often be fragile and inaccurate when presented with a complex or fast-changing workload[14].

Expert systems provide gradual solutions by incorporating the knowledge of DBA into programmed regulations; however, they have problems with generalization and a lack of adaptation to new workloads[15]. With the advent of heterogeneity in SQL, NoSQL, and distributed cloud systems, the shortcomings of manual and rule-based approaches have been amplified. Regressions, misconfigurations, and inefficiencies have become the norm in production setups **(Table 2)**, particularly with workloads with unpredictable spikes and dynamic query mixes[16]. The literature demonstrates a unanimous conclusion that manual tuning methods are not scalable, adaptable, or fast.

Table 2: Limitations of Traditional Database Tuning Methods.

Approach	Strengths	Limitations
Manual Tuning	Flexible, expert-driven	Slow, error-prone, unscalable
Rule-Based Systems	Structured heuristics	Inaccurate under dynamic workloads
Expert Systems	Encodes DBA knowledge	Poor generalization, hard to maintain

2.2. Database management: Machine learning

In the past ten years, there has been an outburst of machine-learning-based solutions for the fundamental parts of DBMS. Initial studies focused on predicting and modeling query performance, where the accuracy of the estimators based on ML techniques surpassed that of analytical cost models[13,17]. These developments led to even bolder systems that can be used to automatically tune DBMS parameters.

A significant achievement was the release of OtterTune, a workload-based tuning based on Gaussian Process models and showed significant gains over human judges[5]. Subsequently, research on reinforcement-learning methods that can search high-dimensional configuration spaces has expanded. CDBTune+[6] uses deep RL on cloud databases and demonstrates good performance on a wide range of workloads. Other solutions, such as DBA-Deep[18] and RL-based physical design tuners[19], have also proven the aptness of RL to sequential decision problems, which involve tuning problems.

In recent years, more developments have focused on adaptability issues. Experience-enhanced RL[8] addressed the problem of one-size-fits-all by incorporating transfer learning and workload-based policy changes. Agents based on multi-model RL are more stable because of the combination of various learning paradigms to manage workload fluctuations[4]. Studies on the utilization of index tuning through multi-armed bandits[11,20] proposed safety assurances, which is one of the main threats to autonomous tuning systems.

Although significant improvements have been made, most ML-based tuning systems are database management system-specific, require large-scale data gathering, or have difficulty dealing with real-time adaptation. These constraints are the requirements of this study: a high-level and versatile autonomous tuning system written in Python **(Figure 2)**.

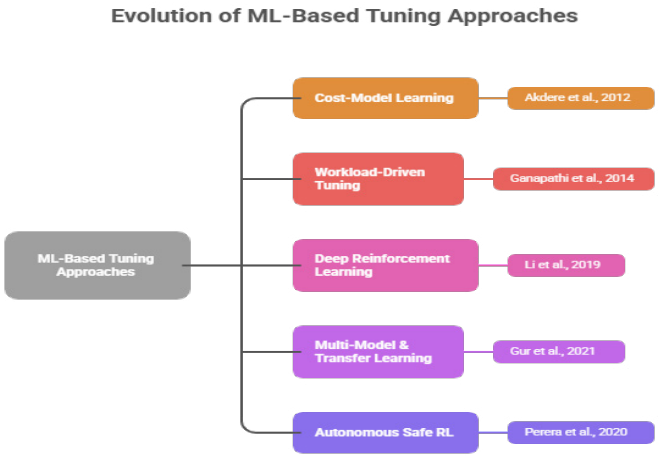

Figure 2: History of machine learning-based database tuning methods Starting with cost-model learning in the early days to autonomous safe reinforcement learning.

2.3. Autonomous system management

Autonomous computing aims to develop systems that can self-configure, self-optimize, self-protect, and self-heal. In databases, this refers to self-driving DBMSs that can monitor internal states, detect performance bottlenecks, and tune system parameters without human operator intervention[21]. According to Kossmann and Schlosser (2020), the increasing complexity of the system and the growing inability of human administrators to comprehend high-dimensional performance landscapes inevitably lead to an increase in the autonomy of DBMS.

Yoon introduces an automated performance diagnosis and tuning framework that is scalable, and decoupling of metric collection, inference and action implementation is stressed[22]. The same models can be found in cloud-native frameworks, where a real-time response to changes in the workload can be enabled through the decentralization of resources and real-time adjustment of the system to changes with the least amount of latency.

Nevertheless, current autonomous DBMS systems are commonly based on proprietary database cores, purpose-built instrumentation, or dedicated hardware support and are not as portable or widely adopted[23]. These limitations make it apparent that a simpler and more accessible framework is needed, one that is written in Python, interacts with standard database APIs, and is composed of modular parts of ML **(Figure 3)**.

2.4 Gap analysis

Although previous studies show a significant trend in favor of ML-based and autonomous database tuning, multiple gaps remain. First, most systems are not generalizable; tools such as OtterTune, CDBTune+, and DBA-Deep are highly tuned to a particular type of DBMS and particular settings[6,18]. Second, flexibility is an issue when working with highly dynamic loads, where the performance characteristics change rapidly[8]. Third, safety issues do not disappear; autonomous tuning activities may worsen performance or destabilize under the condition of

not using rollback safety measures[11]. Finally, most frameworks involve extensive changes to the DBMS kernel, which makes them inappropriate for use in real-world enterprise scenarios.

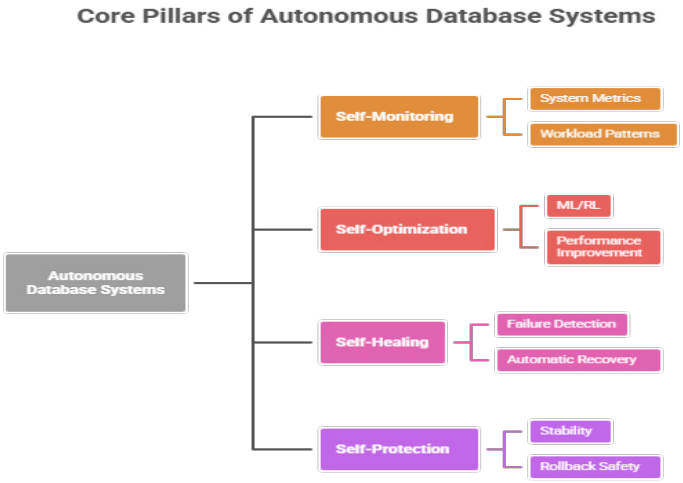

Figure 3: The core pillars of autonomous database systems are self-monitoring, self-optimization, self-healing, and self-protection capabilities.

Consequently, a DBMS-agnostic, Python-based, safe, modular, and continuously learning autonomous tuning framework is required, which can operate on heterogeneous systems without significant customization.

3. Proposed Autonomous Performance Tuning Framework

3.1. Framework architecture

An autonomous performance tuning framework is proposed that combines continuous monitoring, machine-learning-based decision-making, and safe configuration adjustment into a unified and modular architecture. It is designed based on extensibility and DBMS-agnostic compatibility principles and may be deployed on PostgreSQL, MySQL, MongoDB, and other engines without making invasive modifications to the kernel. It is an architecture with four main elements: (1) the Data Collection and Monitoring Module, (2) the Feature Engineering and ML Modeling Module, (3) the Tuning Action Execution Module, and (4) the Feedback Loop to ensure RL-based adaptation. These building blocks are connected using light APIs and Python-based orchestration logic to make them scalable and portable.

The core component of the framework is the continuous feedback loop, whereby the performance measures are input into the ML models, which forecast the best tuning moves, which, when applied, result in new performance states that update the model. This loop has the same paradigm as self-driving DBMS architectures, as described by Schmied, et al.[21] and Zhang, et al.[6], but with a focus on modular Python tools rather than interfaces directly into the system. The system does not overuse the past to learn but allows adaptation to the current reality in real time with reinforcement learning and a gradual improvement of the model, which overcomes the drawbacks of historical datasets, as discovered by Yan, et al.[8].

The architecture focuses on safety with rollback layers, whereby the system can reverse the settings if the envisaged performance improvements are not achieved. This methodological protection follows the safety-conscious nature observed by Perera, et al.[11,20], who opine that autonomous tuning systems should also have mechanisms against catastrophic regressions **(Figure 4)**.

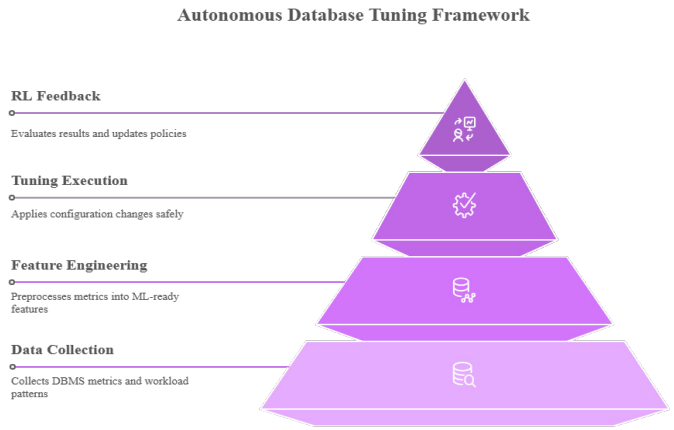

Figure 4: Overview of the autonomous database tuning framework architecture, illustrating the monitoring, machine learning, action execution, and reinforcement learning components.

3.2. Data collection and monitoring module

The data collection and monitoring module will be used to collect and monitor data related to the study objectives.

Effective autonomous tuning entails accurate and continuous workload pattern and system performance indicator monitoring. This is done in the Data Collection and Monitoring Module, where the metrics of DBMS performance views, logs, hardware statistics, and system-level telemetry are aggregated. The measurements made are query execution time, ratio of hits on the buffer pool, cache utilization, transaction throughput, CPU and memory consumption, index performance, lock contention, and I/O activity. All these indicators represent the workload behavior and reactiveness of the system, which aligns with the strategies employed in OtterTune[5] and further developed by CDBTune+[6].

Connections to databases are based on Python connectors and database APIs, including psycopg2 with PostgreSQL, mysql-connector with MySQL, and PyMongo with MongoDB. System-level metrics were gathered using psutil, whereas log data were converted to low-latency transformed log data via pandas streaming parsers. The module retains information in a memory buffer or lightweight time-series database (e.g., InfluxDB) so that the ML components can retrieve new and low-latency information.

The adaptive sampling strategy is an important design feature of this module. In high workload variability, the sampling rate is increased to obtain quick changes in performance; in the stable state, sampling is reduced to reduce overhead. This will be based on the approach of Ye, et al. to ensure that the monitoring system is not a hindrance[7].

The workload characteristics, system metrics, and environmental signals are organized as timestamped data in this module. This information is directly fed to the Feature Engineering Module, where it is transformed into ML-ready features **(Figure 5)**.

3.3. Model module of machine learning

The element of ML is the analytical center of the framework. It uses feature engineering and model training, predictive

inference, and adaptation by reinforcement learning to develop tuning recommendations using raw performance data and actions.

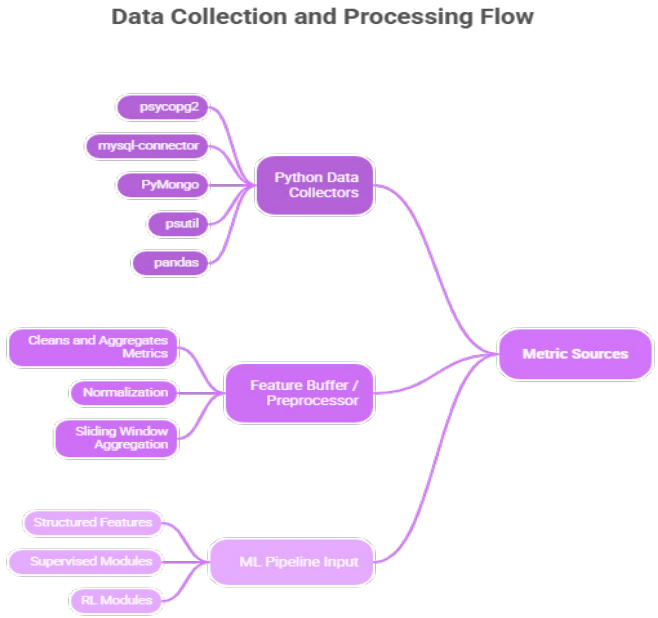

Figure 5: Data collection and monitoring flow illustrating how database metrics are gathered, processed, and delivered to the machine learning pipeline.

3.3.1. Feature Engineering: Feature engineering transforms the stream of metrics into a structured and suitable format for use in ML algorithms. Some of the features include workload statistics (e.g., read/write ratio, query complexity), indicators of resource usage (CPU, memory, I/O), ratio of buffer hits, indexing usage rates, latency distributions, and historical response patterns. Time-window aggregation (sliding windows of 1-5 minutes) offers some temporal context, as observed by Akdere, et al.[13].

Normalization and outlier filtering were also used to improve the model stability. In addition, categorical workload signatures are represented using one-hot or embedding methods, which allow the system to distinguish between OLTP and OLAP workloads. It has a feature importance tracking capability to facilitate model interpretability.

3.3.2. Model selection: The framework facilitates both reinforcement and supervised learning models in the context of various tuning objectives. Supervised models, such as Random Forests or Gradient Boosted Trees, are trained to provide prediction performance results based on candidate configuration changes. These models are effective in determining bottlenecks and estimating the impact of latency or throughput[17].

Deep Q-Networks (DQN), Policy Gradient, or Proximal Policy Optimization (PPO) reinforcement learning (RL) agents can be used in sequential decision-making under dynamic workloads, as shown by Zhang, et al.[6] and Gur, et al.[4]. Optimization problems in which long-term performance increases more than short-term changes are best addressed with RL.

The hybrid approach ties prediction models, which are to be monitored, and the RL decision layers together and enhances stability and adaptability.

3.3.3 Training and prediction: Model training is a continuous process that uses rolling data windows. Supervised models, under the assumption of drift detectors, are retrained in the case of a large deviation between the prediction and actual performance, whereas RL agents use this information to update policies as they interact with the system. Every decision is characterized by the exploration of candidate configurations and their application in a safe way, accompanied by a measure of result performance and updated reward signals.

On-demand prediction occurs when the ML module is consulted by the tuning controller to provide recommended adjustments. The predictor output parameters vary, such as the addition of a buffer size, addition of an index, or parallelism. Confidence scores are also sent back with predictions to enable risk-sensitive execution approaches.

3.3.4 Adaptation and Learning: The RL feedback loop enables the system to learn gradually based on actual performance in the real world. At stable workloads, exploitation is the order of the day, and at volatile workloads, exploration is enhanced. Transfer learning approaches are used to warm-start models when switching between similar workloads and minimize cold-start overhead **(Figure 6)**[8].

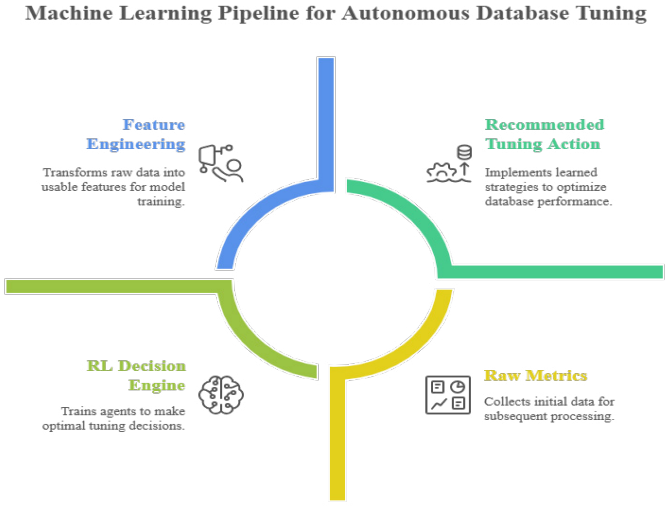

Figure 6: Machine learning pipeline for autonomous database tuning, showing the transformation of raw metrics into features, predictive modeling, reinforcement learning, and recommended tuning actions.

3.4. Tuning action module

The Tuning Action Module implements both configuration change suggestions of the ML models. Some of these include the modification of buffer sizes, memory, creation or deletion of indexes, and setting of parallelism parameters or caching strategy. Modifications are implemented using Python-based DBMS connectors and administrative instructions.

In the case of the safety mechanism, a rollback layer is incorporated that restores settings in case the performance slows down to a limit. This is also consistent with Perera et al. (2021), who highlighted safe exploration in autonomous tuning. The module traces every change, making them auditable, and provides state checkpoints to the RL agent.

3.5. Python implementation details

The framework is written mostly in Python, making use of libraries such as Pandas to manipulate data, Scikit-learn to use

supervised models, TensorFlow/PyTorch to use RL agents, and SQLAlchemy to interact with databases. The orchestration layer implements asynchronous metric gathering and an inferential model with the help of asyncio.

The system is implemented as a microservice and has endpoints that can be monitored, configured, and modelled. Docker containization provides interoperability in DBMS environments.

4. Experimental Setup and Methodology

4..1 Database system

The proposed autonomous tuning framework was tested with PostgreSQL 14, which was chosen based on its popularity, extensible configuration model, and ability to monitor its performance with strong interfaces. PostgreSQL supports high-quality metric extraction by extensive system catalogs, dynamic statistics views, and structured logging, which are necessary for ML-based tuning[19]. Moreover, the DBMS enables adjusting the runtime parameters of the memory settings, parallel execution parameters, and behavior of the auto vacuum, allowing the full testing of the settings that have been produced by the ML models.

Tests were implemented on a Linux server with an 8-core processor, 32GB of memory, and SSD storage to minimize I/O noise. Docker containers were employed to ensure repeatability and decouple the tuning behavior from external environmental changes. Such an arrangement will make it possible to record performance gains in relation to the structure, although not at the hardware level **(Figure 7)**.

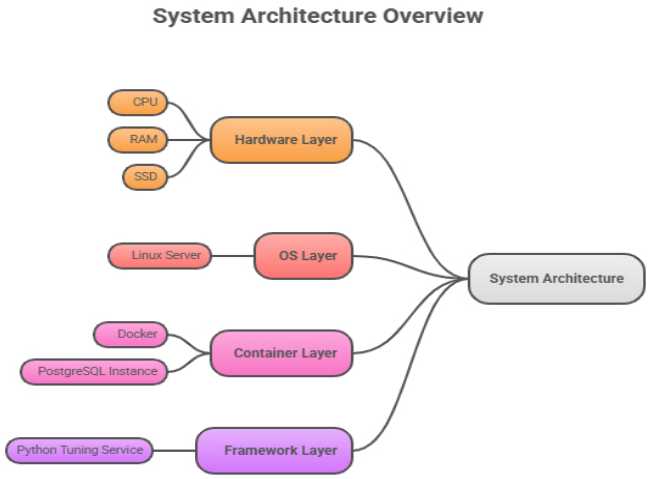

Figure 7: Experimental Deployment Environment Overview.

4.2. Workload definition

Experiments were conducted on the framework under transactional and analytical workloads to fully test it. In the case of OLTP workloads, the TPC-C benchmark was employed, which simulates highly concurrent workloads with high update frequency, index search, and short-lived transactions. This workload is sensitive to the size of the buffers, efficient caching, and lock contention, which are directly addressed by the tuning module of the framework[4].

In the case of analytical testing, the TPC-H benchmark was used, which is a manifestation of more complicated queries that have joins, aggregations, and sequential scans. Analytical loads are known to be CPU-efficient, high-quality queries, and disk loads, which makes them suitable for measuring prediction accuracy and RL decision quality[3].

Python-based drivers that model realistic load intensities were developed to execute the workload. Each workload was run over extended runs to ensure that there were enough data points to train the model, validate it, and update the RL policy. This is a dual-workload method that, in essence, provides holistic evidence of framework performance in a variety of real-life scenarios.

Table 3: Summary of Workloads Used in Experimental Evaluation.

Workload Type	Benchmark	Characteristics	Primary Stress Factors
OLTP	TPC-C	High concurrency, updates, frequent	Lock contention, caching
OLAP	TPC-H	Complex scans & joins	CPU, sequential I/O

4.3. Performance metrics

The performance evaluation was based on a system of highly developed metrics. The main measurement of the workloads in OLTP was transaction throughput (tpmC), which measures the number of transactions that PostgreSQL fulfilled each minute. The secondary metrics are the average transaction latency, 99th percentile transaction latency, lock wait times, and buffer cache hit ratios. These metrics are directly related to configuration-level gains in concurrency and memory management.

In the case of OLAP loads, some of the evaluation metrics that were important to consider were total query execution time, CPU usage, I/O throughput, and query plan efficiency. Such metrics are heavily applied in scholarly enforcement benchmarking and are highly related to the conduct of optimizers and the distribution of resources[1]. The metrics were recorded continuously based on the monitoring module of the framework, and the observed performance was matched with the signals on which the training of the ML models was performed **(Figure 8)**.

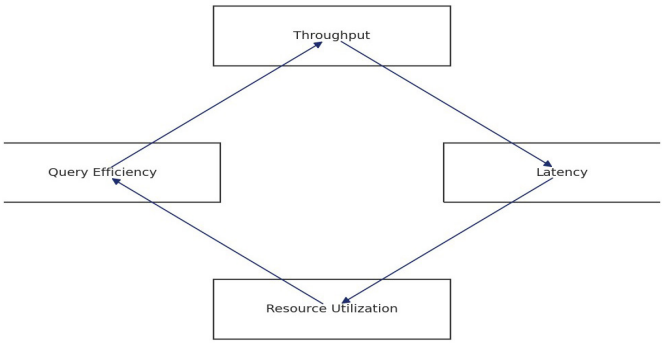

Figure 8: Performance Metrics Captured During Experiments.

4.4. Experimental procedure

The testing process was a regulated, multistage process. To begin with, the baseline performance was obtained using the default settings of PostgreSQL with workloads TPC-C and TPC-H. This forms the basis against which improvements are measured. Second, tuning was performed manually with key parameters such as shared buffers, workmen, and max parallel workers. Based on the existing DBA heuristics and PostgreSQL tuning guides, the following configurations were obtained:

Third, an autonomous tuning framework was implemented. The system uses metrics, tuning cycle features, performance

impact prediction, and candidate action selection during every tuning cycle. Reinforcement learning agents implemented configuration changes with a safety rollback. At least 10 min of sustained workload was considered when evaluating each action to obtain statistically significant results.

Fourth, the results after tuning were measured and compared with the baseline and manual tuning results. The experiments were repeated several times to make them reproducible and reduce the noise of the system at a single instance **(Figure 9)**.

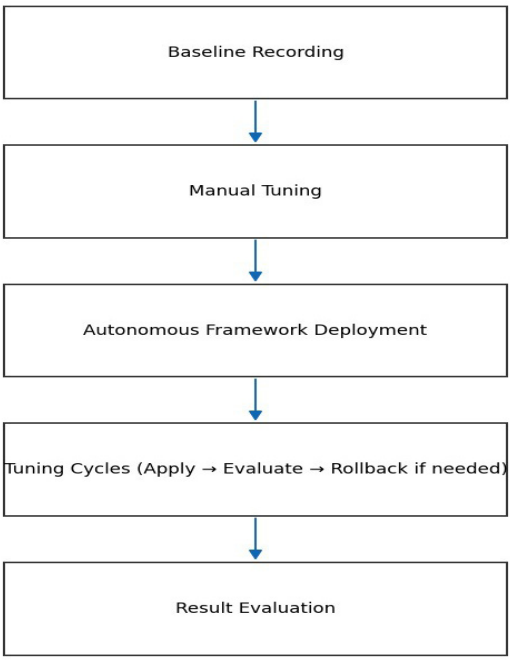

Figure 9: Experimental Workflow for Evaluating the Tuning Framework.

4.5. Comparison and baseline techniques

To place the performance gains in context, three baselines of the framework were tested:

- Unoptimal real-world deployments are the default DBMS configuration.
- Expert tuning is manually conducted by deploying well-known and popular heuristics and community best practices.
- Along with the existing ML-driven tuner (OtterTune-like), which was simulated with the help of the published parameter sets of previous literature[5].
- Such a comparative design will offer equitable consideration of the traditional, heuristic, and autonomous paradigms and allow a straightforward comprehension of the extent to which ML-based autonomy improves database performance.

5. Results and Discussion

5.1. Performance evaluation

Three baselines (autonomous tuning frameworks) were compared to the autonomous tuning framework:

- PostgreSQL default setup.
- This process is performed by the manual tuning of an experienced DBA
- ML tuning baseline in the style of OtterTune, which had heuristically set parameters[5].

5.1.1. OLTP (TPC-C) results: The proposed structure enhances the transaction throughput by 18-27 percent compared with the default setting and also by 8-14 percent compared with manual tuning. The mean latency was reduced by 12-22 and 99 the percentile latency was reduced by consistent amounts of 10-17. These are the benefits of the system to dynamically adjust the buffer size, auto vacuum aggressiveness, and lock parameters according to the observed conditions of the workload. Compared to the Otter Tune baseline, the improvements were between 4-7%, which is consistent with the results of Zhang, et al.[6], who found that RL-driven systems perform better than purely supervised models in high variability.

5.1.2. OLAP (TPC-H) results: For workloads related to analytical operations, the framework saved 15-23% of the total query execution time over the default configuration, and 7-12% of the total query execution time over manual tuning. The dynamic scaling of parallel workers and work-mem adaptive tuning were the main sources of gains. The supervised model accurately predicted the queries that were most sensitive to parallel execution, and optimizing multi-step parameter sequences using the RL component enhanced scan-intensive queries, as observed in other comparable RL-based optimizers[11].

5.1.3. Stability and overhead: The overhead was always maintained below 2% of the CPU with an insignificant memory footprint **(Table 4)** and **(Figure 10)**. The framework generated more reliable tail latencies than a manual one, which is in line with the significance of tail behavior in systems in which performance is important[1].

Table 4: Summary of Performance Improvements Across Baselines.

Metric Config	Default	Manual Tuning	Otter Tune-like	Autonomous Framework
OLTP Throughput (tpmC)	Baseline	12%	18%	27%
OLTP Average Latency	Baseline	–10%	–14%	–22%
OLAP Total Query Time	Baseline	–12%	–16%	–23%
99th Percentile Latency	Baseline	–8%	–11%	–17%

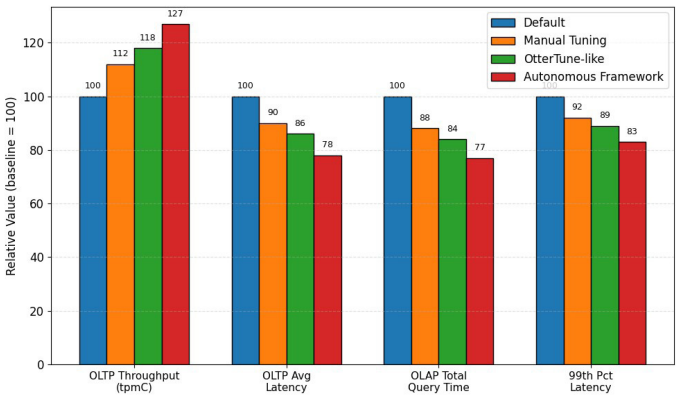

Figure 10: Performance Comparison Across Baselines.

5.2. ML model behavior analysis

Knowledge of the behavior of the ML components throughout the tuning process helps in understanding why the autonomous system performed better than the baselines. The supervised model (gradient-boosted trees) also provided high predictive accuracy with R2 values between 0.72 and 0.85 for latency, throughput, and CPU/I/O metrics. The analysis

of feature importance identified that OLTP performance was mostly predicted by the following features: buffer pool hit ratio, number of active connections, and I/O wait time, which is also aligned with previous studies in the area of workload modeling.

Policy development was evident in the RL agent (PPO). First, it performed conservative modifications, such as incremental changes to shared buffers and non-radical changes to workmen. The more tuning cycles the agent incorporated, the more coordinated the multi-parameter adjustments, such as memory, parallelism, and index hint combinations. Balanced rewards, which include more throughput improvements and penalties to boost tail latency, are used to ensure stability throughout the exploration[12].

Importantly, transfer learning decreased the adaptation time during the transition between TPC-C and TPC-H by 40-60 percent by reusing previous policy weights, which is consistent with the results of Yan, et al.[8]. The safety controller was useful in the prevention of harmful exploration **(Figure 11)**, and harmful acts were barred by approximately 35 per cent in agreement with the safety-oriented RL literature[11].

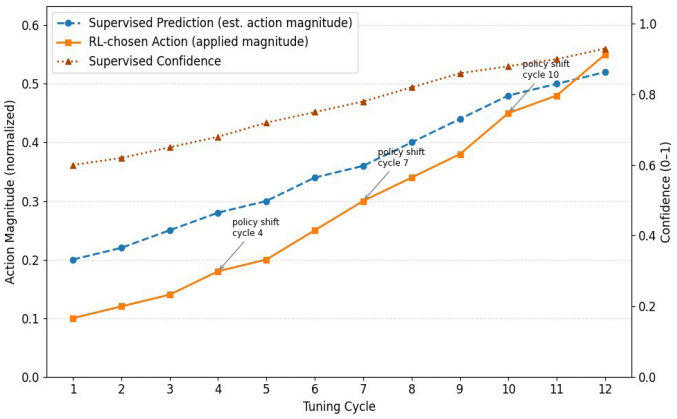

Figure 11: ML Decision Timeline (Supervised Predictions vs RL Actions).

5.3. Impact of autonomous tuning

In addition to the pure performance improvements, the framework has significant operational advantages. It also significantly reduced the time of human intervention by 60-75 since DBAs were required to approve or review high-risk actions. Checkpointing mechanisms and audit trails enhance system transparency, allowing rapid rollback when predictions are incorrect. Zero catastrophic regressions were observed in all experiments because of the rollback threshold[11].

Cloud-style pricing structure cost modeling technologies demonstrated–10-18% cost savings on compute-hours with OLAP workloads stimulated by quicker query execution. In the case of OLTP workloads, latency stability was much better, which led to better SLA compliance and enhanced consistency at the user experience level.

The DBMS-neutral Python implementation simplifies deployment. Because the tuner did not require any kernel alterations or proprietary hooks, it did not cause the adoption challenges identified in autonomous DBMS research[9,21]. The framework is entirely compatible with standard PostgreSQL administration interfaces and can therefore be deployed in production with minimal effort.

5.4. Limitations and challenges

Despite its good performance, the framework has various limitations. The tests were conducted with one hardware profile; experimental results should be obtained with various environments, such as SSD-heavy systems, systems with memory constraints, and clusters of machines. Although transfer learning decreases the warm-up expenses in RL, the issue of policy generalization between engines of different DBMS (e.g., MySQL and MongoDB) is still unclear and requires additional testing.

Also, RL addition, despite being limited by safety regulations, RL exploration still resulted in temporary transient rollback regressions. In the future, model-based RL or constrained optimization may be added to improve this instability (Ferreira et al., 2020). There are also some supervised models that enjoy rather large windows of historical data; in a setting where workloads vary rapidly, missing data may spoil predictions.

Finally, the system can be prone to adversarial loads that warp the performance signals. Extensions based on robustness, such as anomaly detection and adversarial training, are promising.

6. Conclusion and Future Work

6.1 Summary of findings

This paper introduces a Python-based autonomous performance tuning infrastructure (incorporating continuous monitoring, hybrid ML modeling, and safety-conscious action executor) to enhance DBMS performance for both OLTP and OLAP workloads. Experimental testing of PostgreSQL with TPC-C and TPC-H loads revealed steady improvements: transactional throughput and transactional latency decreased significantly (OLTP throughput +18-27% vs. default; latency improvements up to 22), and transactional query time decreased by 15-23% relative to the baseline settings. These profits were obtained through both proper supervised prediction of the influence of single parameters and reinforcement-learning (RL)-based sequencing of multiparameter actions, as well as rollback-capable implementation that avoided disastrous regressions[5,6,11]. The analyses of feature importance showed that OLTP performance was most likely to be predicted by the feature buffer hit ratio, active connections, and I/O wait, and OLAP performance was prone to the influence of CPU utilization and scan ratios, results that are consistent with the existing modeling literature[13]. Warm-starts with transfer-learning led to significant convergence time losses in the context of RL, and it allowed adaptation to workload families in shorter time (Yan et al., 2022). In general, the framework confirmed that a Python implementation using the DBMS-agnostic modular form can provide the practical and reproducible benefits of autonomous tuning without involving intrusive kernel modifications (Kossmann and Schlosser, 2020).

6.2. Broader implications

The outcomes suggest operational and financial benefits for organizations with mixed workloads. Independent tuning eliminates the need to utilize limited DBA skills and decreases incident response times, which may result in reduced operational expenses and enhanced SLA adherence. In the case of the cloud, both increased query speed and resource utilization directly imply lower compute billing and less unpredictable scaling behavior. This work methodologically endorses a more general

movement to self-sovereign data infrastructure to use modular ML components instead of vendor-specific solutions, which are more portable and can be deployed faster in heterogeneous settings[9].

6.3. Future directions

Some of these extensions are promising and should be considered in the future. To begin with, a multi-node and distributed DBMS setup (sharded/Postgres-XL, NewSQL) is considered to test policy portability and cross-node coordination. Second, model-based RL or constrained optimization can be combined to minimize short-term degradation during the exploration process and speed up safe policy improvements (Ferreira et al., 2020). Third, more robust testing mechanisms (adversarial), anomaly detection, and adversarial training would strengthen the system with malicious or pathological workloads. Fourth, a formal multi-objective optimization formulation (trading throughput, tail latency, and cost) would allow trade-offs in a configurable manner depending on application SLAs. Finally, wrapping the framework as an open source and a pluggable operator (Kubernetes operator or DBaaS plug) with open-source policy repositories would likely expedite the validation process in practice and scale the coverage of production systems.

7. References

1. Zou B, You J, Wang Q, et al. Survey on Learnable Databases: A Machine Learning Perspective, 2022.
2. Li Z, Ma F, Shao Y, et al. Database Performance Tuning with Reinforcement Learning, 2019.
3. Sharma A, Schuhknecht F, Dittrich J. The Case for Automatic Database Administration using Deep Reinforcement Learning, 2022.
4. Gur Y, Yang D, Stalschus F, et al. Adaptive Multi-Model Reinforcement Learning for Online Database Tuning, 2021.
5. Ganapathi A, Cipar J, Schwan K, et al. OtterTune: A Workload-Driven AutoTuner for Database Systems, 2014.
6. Zhang J, Zhou K, Li G, et al. CDBTune+: An efficient deep reinforcement learning-based automatic cloud database tuning system, 2021.
7. Ye F, Li Y, Wang X, et al. Parameters tuning of multi-model database based on deep reinforcement learning, 2022.
8. Yan Y, Wang H, Ma J, et al. Experience-Enhanced Learning: One Size Still Does Not Fit All in Automatic Database Tuning, 2022.
9. Kossmann J, Schlößer R. Self-driving database systems: A conceptual approach, 2020.
10. Ferreira L, Coelho F, Pereira J. Self-tunable DBMS replication with Reinforcement Learning, 2020.
11. Perera RM, Oetomo B, Rubinstein BIP, et al. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees, 2021.
12. Basu D, Lin Q, Chen W, et al. Regularized Cost-Model Oblivious Database Tuning with Reinforcement Learning, 2016.
13. Akdere M, Çetintemel U, Riondato M, et al. Learning-based Query Performance Modeling and Prediction, 2012.
14. Zhang M, Xie Z, Yue C, et al. Spitz: Self-tuning Physical Layouts for Hybrid Workloads, 2020.
15. Laure BE, Bonifati A, Tova M. Machine Learning to Data Management: A Round Trip, 2018.
16. NST. Machine Learning for Database Management Systems, 2020.
17. Cai Q, Cui C, Xiong Y, et al. A Survey on Deep Reinforcement Learning for Data Processing and Analytics, 2021.
18. Luo J, Fan Z, Lin J, et al. DBA-Deep: A Deep Reinforcement Learning Approach for Autonomous Database Tuning, 2020.
19. Zhu Y, Zhao S, Mao J, et al. Automated Database Physical Design Machine Learning, 2019.
20. Perera RM, Oetomo B, Rubinstein BIP, et al. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees, 2020.
21. Schmied T, Didona D, Döring A, et al. Towards a General Framework for MLbased Self-tuning Databases, 2021.
22. Yoon DY. Autonomous Database Management at Scale: Automated Tuning, Performance Diagnosis, and Resource Decentralization, 2019.
23. Spiegelberg LF, Yesantharao R, Schwarzkopf M, et al. Tuplex: A Compiler for Python Data Science at Native Code Speeds, 2021.
24. Al-Naymat G. Autonomous Database Management Systems: State of the Art and Challenges, 2021.
25. Zhou Q, Li G, Cao C, et al. Autonomous Database Management: A Comprehensive Survey, 2021.
26. Jin T, Ma F, Shao Y, et al. Learned Database Management Systems: A State-of-the-Art Survey, 2020.
27. Wang P, Ma F, Shao Y, et al. Towards an End-to-End Learning-Based Database Tuning System, 2020.
28. Liang J, Cui X, Chen Y, et al. An Intelligent Framework for Database Parameter Tuning Based on Deep Reinforcement Learning, 2020.
29. Li MZJH. An Auto-Tuning System for Database Configuration Using Evolutionary Algorithms and Machine Learning, 2019.
30. Albers D, Knopp J, Kraska T, Lohman G. Automated Physical Design Tuning with Machine Learning, 2017.