Vista: Machine Learning based Database Performance Troubleshooting Framework in Amazon RDS

Vikramank Singh
Amazon Web Services
vkramas@amazon.com

Zhao Song[†]
Amazon Web Services
songzhao@amazon.com

Balakrishnan (Murali) Narayanaswamy
Amazon Web Services
muralibn@amazon.com

Kapil Eknath Vaidya
Amazon Web Services
kapilvaidya24@gmail.com

Tim Kraska
Amazon Web Services, MIT
kraska@mit.edu

ABSTRACT

Database performance troubleshooting is a complex multi-step process that broadly involves three key stages– (a) Detection: determining what's wrong and when; (b) Root Cause Analysis (RCA): reasoning about why is the performance poor; (c) Resolution: identifying a fix. A plethora of techniques exist to address each of these problems, but they hardly work in real-world at scale. First, real-world customer workloads are noisy, non-stationary and quasi-periodic in nature rendering traditional detectors ineffective. Second, real-world production databases execute a highly diverse set of queries that skew the database statistics into long-tail distributions causing traditional RCA methods to fail. Third, these databases typically execute millions of such diverse queries every minute rendering traditional methods inefficient when deployed at scale.

In this paper we describe `Vista`, a machine learning based performance troubleshooting framework for databases, and dive-deep into how it addresses the 3 real-world problems outlined above. `Vista` deploys a deep auto-regressive model trained on a large and diverse Amazon Relational Database Service (RDS) fleet with custom skip connections and periodicity alignment features to model long range and varying periodicity in customer workloads, and detects performance bottlenecks in the form of outliers. Furthermore, it efficiently filters only a top few dominating SQL queries from millions in a problematic workload, and uses a robust causal inference framework to identify the culprit queries and their statistics leading to a low false-positive and false-negative rate. Currently, `Vista` runs on hundreds of thousands of RDS databases, analyzes millions of workloads every day bringing down the troubleshooting time for RDS customers from hours to seconds. At the end, we also describe several challenges and learnings from implementing and deploying `Vista` at Amazon scale.

CCS CONCEPTS

• **Information Systems → Database Management Systems**.

KEYWORDS

Cloud Databases, Performance Troubleshooting, ML for Systems

ACM Reference Format:
Vikramank Singh, Zhao Song[†], Balakrishnan (Murali) Narayanaswamy, Kapil Eknath Vaidya, and Tim Kraska. 2024. `Vista`: Machine Learning based Database Performance Troubleshooting Framework in Amazon RDS. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3698038.3698519

1 INTRODUCTION

Relational databases are at the center of most fundamental business processes, and poor database performance impacts both revenue and customer experience. A performance bottleneck in database can occur due to a large number of reasons, for e.g., concurrent clients competing for the same resources (e.g., CPU, I/O), in combination with the infrastructure or network failures. Owing to the importance of monitoring performance, databases constantly collect a large number of detailed telemetry [51] (e.g., MySQL collects over 260 different metrics). However, when a performance bottlenecks occurs, it is challenging and time consuming for database administrators (DBAs) to monitor all these metrics,

[†]Work done while at AWS.

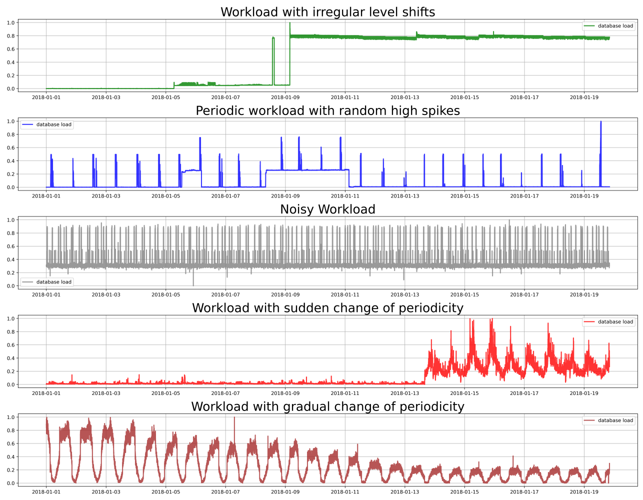


Figure 1: Real workloads are noisy, non-stationary, and quasi-periodic in nature making it non-trivial to identify when to alert the customer about a performance degradation.

analyze millions of heterogeneous queries, and identify the root-cause in real-time. As modern applications get increasingly deployed on the cloud platforms using database-as-a-service, DBAs rely on the cloud providers to provide them with the necessary debugging framework to help detect, diagnose and fix such bottlenecks in real-time.

A typical performance debugging framework has three key components: (a) *Detection*: to detect performance degradation using telemetry data; (b) *RCA*: to identify the root cause in form of either a problematic query, sub-optimal configuration, over or under-provisioned hardware, etc; (c) *Recommendation*: provide necessary information on how to fix the identified issue (e.g., in forms of informative troubleshooting docs, query tuning hints, etc). In order for such a performance debugging system to be deployed in production, it needs to satisfy some key requirements.

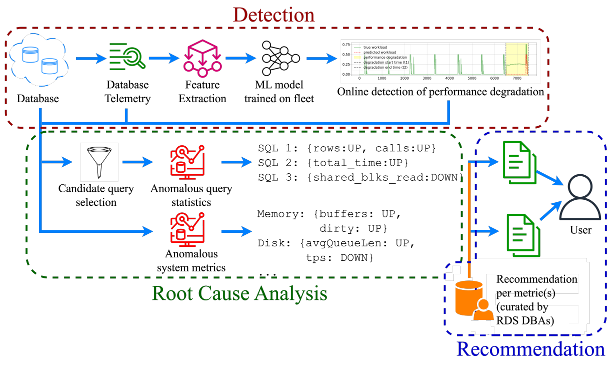
First, the detection component should work for variety of database workloads across the fleet. For a fleet of millions of databases, it is infeasible to build and maintain local detection models for each individual database. In Figure 1 we show 5 real customer workloads, measured over a period of 21 days, represented in form of number of active sessions per minute, and normalized for better visualization. Workloads differ significantly from one another making it difficult to pick the right detection algorithm (e.g., should we optimize for detecting level-shifts, or periodic spikes, or point outliers?). To make matters worse, workload behavior across a single database is also dynamic.

Second, the RCA component of the system should be robust to noise in the workload. RCA is generally performed by comparing the query statistics or system metrics in the anomalous regions with baseline regions in history [29, 51]. Here identifying the right baseline is key to good precision/recall. In production workload, it is non-trivial to identify the right baseline regions due to changing periodicity, old anomalies, and noise in workload. The RCA component should be robust to such behavior in the historical workload when computing the baseline. In Figure 3 we show the number of executions of a SQL query increased during a detected performance anomaly (red region in top plot). However, simply using the past $\mathcal{H}$ days to compute a baseline statistic (e.g., mean or p90) leads to false negatives. Furthermore, given how diverse real-world workloads are, queries tend to have a long-tail distribution as seen in this example which prevents us from deploying popular statistical techniques (e.g., T-test [40]) in production that most of prior works rely upon [29, 51].

Past Research: The area of performance troubleshooting is relatively well studied in the systems community and hence there exist tons of solutions that address different aspects of database performance troubleshooting, for e.g., detection of performance bottlenecks [20, 33, 51], root causing analysis [23, 28–30], and recommendation in form of database tuning or query optimization [12, 18, 27, 31, 32, 38, 39, 48, 52]. Although insightful, most of these ideas are rendered ineffective or inefficient when translated directly to real-world customer workloads.

The detection techniques do not work due to various simplifying assumptions made about the data (e.g., stationarity in data [29, 33], gaussian noise [51], access to labeled dataset [19], etc), or their infeasibilty to run at scale (e.g., require human labeling [51], access to customer workload for replay [48], etc).

Existing RCA techniques rely on classical statistical tests like T-test [51], Mann-Whitney [33], or correlation (e.g., Spearman's coefficient[29]) to identify abnormal features of SQL queries (e.g., query statistics, or metrics like CPU, Memory, I/O, etc). Although interpretable, these tests make simplistic assumptions about the data that do not hold in real world. For e.g., DBSherlock [51] constructs predicate-based illustrations of anomalous metrics by comparing the arithmetic `mean` of *'normal'* and *'anomalous'* time regions. Figure 7, and 8, show how comparing `mean` can be misleading in presence of long-tail distributions. Complex techniques like CauseRank [30] propose learning a causal graph over anomalous metrics. Translating this idea to anomalous queries is non-trivial as production queries are diverse, rare, and ephemeral making it hard to train any feasible model. Also, production systems encounter millions of queries during peak hours. Maintaining such a graph with non-stationary query workload is infeasible at production scale.


Detection
Database
Database Telemetry
Feature Extraction
ML model trained on fleet
Online detection of performance degradation
Candidate query selection
Anomalous query statistics
Anomalous system metrics
Root Cause Analysis
User
Recommendation per metric(s) (curated by RDS DBAs)
Recommendation

```
SQL 1: {rows:UP, calls:UP}
SQL 2: {total_time:UP}
SQL 3: {shared_blks_read:DOWN}

Memory: {buffers: UP,
         dirty: UP}
Disk: {avgQueueLen: UP,
       tps: DOWN}
...
```

Figure 2: Vista System Architecture

Production Systems: The industry relies on much simpler light-weight tools (e.g., Performance Insights (PI) [8], Amazon CloudWatch [1], Azure Monitoring [3], Oracle Enterprise Manager [6], etc) which have limited features causing the DBAs to spend hours diagnosing performance issues, inspecting queries manually, and identifying fixes while ignoring hundreds of false alerts. For instance, PI uses simple static thresholds to flag a performance anomaly anytime the number of active sessions exceeds the number of available CPUs. On a noisy production database, this could happen every other minute, leading to thousands of false alerts. In summary, there is clearly a gap between the past research and production systems. SOTA research techniques either assume nicer properties of data, or are non-trivial to scale, and production systems are too simple, and hence to noisy, to rely upon.

We present Vista, an automated machine learning based performance troubleshooting framework for databases built to bridge this gap. The core design philosophy of Vista is: *"How do we use sophisticated techniques that can exploit the vast and diverse RDS fleet to achieve much needed generalization, while deploying them at scale.* Like several other frameworks, Vista too adopts a 3-staged troubleshooting pipeline that consists of- (a) Detection, (b) RCA; (c) Recommendation; which we describe in this paper. However, the focus of this paper are two key modeling insights listed below which also form the contributions:

- Vista shows a robust self-supervised way of training a lightweight deep ML model for detection of performance bottleneck across a fleet of 100K databases that contains diverse set of workloads which vary in periodicity, scale, and type of outliers. This helps the model generalize to variety of unseen workloads.
- Vista shows a robust way of identifying the abnormal queries, and telemetry metrics with heavy-tail distribution for downstream root cause analysis at scale. The main intuition is to ignore noisy regions in the past workload to get a robust baseline, and use quantile estimators to handle heavy-tail behavior.

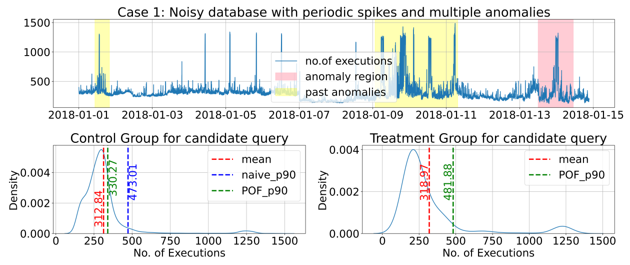


Figure 3: *Naive baselines lead to False Negative (FN):* Standard *mean* (red line), or $p90$ (blue line) are similar across both treatment and control group, indicating that no. of executions of this candidate query didn't change, and hence it isn't a root cause, which is incorrect. Vista computes a robust $p90$ baseline (green line) using POF by excluding occurrences of query during past anomalies (yellow region), and correctly identifies the query as root-cause due to an increase in its no. of executions during the anomaly (red region) w.r.t baseline.

While Vista is not the first ML-based performance troubleshooting framework for databases [29, 33, 51], to the best of our knowledge Vista is the first that highlights the peculiarities of real-world database workloads, demonstrates how to address them using existing machine learning techniques, and deploys at scale. Vista has been running in production[†] at AWS for >2 years making thousands of detection for hundreds of thousands of RDS databases every day. Making Vista practical required several iterations and careful design trade-offs, in particular addressing long-tail distribution of queries, and variance in periodicity across workloads when learning a model across the fleet. We conclude the paper with such lessons learned from implementing and deploying Vista at scale that maybe of interest to those in the "ML for systems" community with an eye for practicality.

2 RELATED WORK

Detection of Performance Degradation: Given the large number of telemetry collected by modern databases, machine learning has been a popular choice for detecting performance bottlenecks [20, 22, 29, 33, 51]. However, 2 key issues make this problem non-trivial are: (a) lack of labeled data, and (b) dynamic workload behavior. Unavailability of labels prevents us from training deeper supervised models, and hence approaches like iSQUAD [33], or PinSQL [29] rely on traditional detection techniques like threshold based outliers or

[†]https://aws.amazon.com/devops-guru/features/devops-guru-for-rds/

T-Test which become ineffective when dealing with long tail distribution or highly dynamic periodic workloads. On other hand approaches like DBSherlock [51] rely on the user to provide a 'normal baseline' so model can identify anomalies with respect to it. This is highly unreliable as identifying the right baseline is a non-trivial task for the end-user both from statistical and domain knowledge perspective. Furthermore, the problem worsens as the workload pattern for a given database changes over time rendering simple models like SR-CNN [42] and STL-based [17, 26, 34, 49] with static thresholds ineffective. `Vista` uses a combination of smart techniques like periodicity alignment and skip connections to handle these highlighted issues without needing any labeled data. We believe `Vista` has the first ML-enhanced performance degradation detection mechanism that can handle highly dynamic periodic workloads of varying lengths trained with zero labels and deployed in a real-world setting at scale.

Root Cause Analysis (RCA): RCA can be stated as identifying the right set of anomalous metrics or SQL queries that can inform the user about 'why' did the performance degrade. DBSherlock [51] generates root-causes in form of predicates whereas Sentinel [20] relies on database logs to build behavior models to identify anomalous metrics. iSQUAD [33] focuses specifically on identifying intermittent slow queries and clusters them into known root causes, while PinSQL [29] identifies SQLs that correlate with performance anomalies. Similarly, CauseRank [30] and FluxRank [28] use causal discovery algorithms to build a causal graph followed by a ranking algorithm to rank the root causes. While these research results show a lot of promise, they do not address the practical issues we faced when deploying `Vista`. For example, DBSherlock constructs predicate-based illustrations of anomalous metrics by comparing the arithmetic means of 'normal' and 'anomalous' time regions. In Figure 7 and 8 we show examples of query statistics collected by PostgreSQL engine for 3 different customer workloads. Simply comparing means of long-tail distributions can lead to large number of both false positives and negatives questioning the reliability of the troubleshooting framework. Similarly, iSQUAD and PinSQL rely on T-test [40], while CauseRank uses z-score, both of which are known to be unstable for long tail distribution, with [28] being a notable exception.

3 OVERVIEW

In this section we describe the high level architecture of `Vista`. All RDS database instances emit hundreds of telemetry metrics every second in a streaming manner. When turned on for a given database instance, `Vista` uses its following components to detect, and analyze a performance bottleneck in real-time and provide recommendation to the user on how to further resolve the issue:

- **Anomaly Detection:** A machine learning model that is responsible for learning the workload behavior over time and detecting outliers.
- **Root Cause Analysis:** A module that smartly filters the dominating queries from millions of queries running every second and uses a robust causal inference framework to identify the root cause or culprits in form of a particular query or system metrics.
- **Recommendation:** A light-weight module that surfaces the technical findings with relevant troubleshooting documentation carefully curated by our expert in-house DBAs.

As stated in Section 1 we will dive deeper into the first two components of `Vista` which also form the core technical contribution of this work. However, at the end we will explain how those two modules are combined with a simple recommendation module to complete the loop and provide a final recommendation to the user.

Figure 2 gives an overview of `Vista`'s system architecture. To understand how `Vista` works, we will briefly walk through the pipeline from start to end. When turned on for a given RDS database instance, `Vista` starts collecting information about all the active sessions on the database every second and feeds it to the feature extractor to compute a surrogate health metric for the database. It then constantly feeds that health metric into a trained ML model in a streaming manner. As soon as an anomaly is detected in the health metric, `Vista` creates an anomaly event with the 4 key attributes–$\{start\_time, end\_time, severity, ongoing\_flag\}$ (we will describe each attribute in detail in section 4 below). If an anomaly is ongoing, it will not have an *end_time*. `Vista` then sends this anomaly event to the *Analysis* module which first identifies the dominating queries as a part of candidate selection, and fetches per-query statistics for those candidate queries. Similarly, it also fetches telemetry data for a subset of important system metrics for that anomalous time region. It then uses a causal inference framework that is robust to long tail behavior and identifies the root cause queries and metrics. Finally, the *Recommendation* module attaches relevant troubleshooting documentation to this statistical analysis. These documents are handcrafted by expert in-house DBAs and contain best practices to follow w.r.t the identified issues. In the `Vista` console, anomalies are ranked w.r.t their *severity* so users can prioritize in case they have to deal with multiple anomalies over a period of time.

4 VISTA

In this section, we dive deeper into the technical details of how `Vista` trains its ML detectors to identify performance

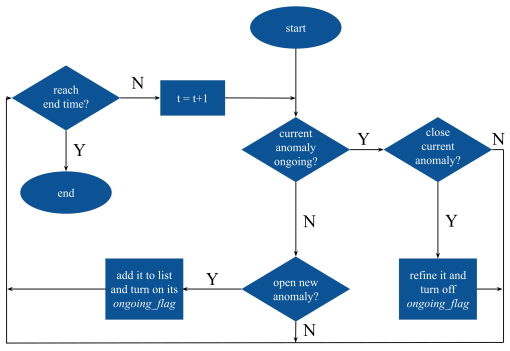


Figure 4: The flowchart of Vista's streaming anomaly detection process.

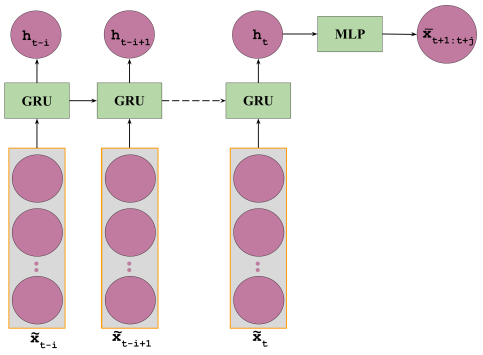


Figure 5: A modified neural network architecture of Vista's forecasting model with skip connections.

bottlenecks in a streaming manner, filters dominating SQL queries, and pin point root causes in presence of long tail distributions.

4.1 Requirements

Based on customer interactions we define a rigorous list of requirements needed to make Vista's detection module practically useful for real-world. First, when turned on, Vista need to identify anomalies in a streaming manner in order to alert customers in real time when a performance degradation occurs. Second, anomalies should be reported in form of continuous time segments with a specific *start_time* and *end_time* along with other attributes instead of point-wise anomalies. This reduces noise both in detection, and RCA processes. Third, customer workloads are highly periodic in nature. To be practically useful, Vista's detection model needs have a low False Positive Rate (FPR) and hence should be able to discount periodic behavior in customer workloads. Fourth, the model needs to be instance optimized, i.e, custom tailored to user's workload. A pre-trained model that has $< 5\%$ FPR for 95% of customers but $> 20\%$ FPR for 5% of customers would not be acceptable. However, given Amazon's scale, it is infeasible to maintain individual models for millions of customers. Hence, Vista's detection model should be able to exploit the large RDS fleet during training and generalize to unseen customer workloads. Finally, sixth, the output needs to be interpretable. This is an important requirement where users demand a single *'health'* metric/score which can indicate the overall state of database system which is easily interpretable to make decisions in time-sensitive scenarios.

To the best of our knowledge, we haven't seen many prior works present such a comprehensive list of requirements coming directly from real customers that makes the detection model practically useful. Each requirement stated above is technically non-trivial, and a research topic in itself valuable enough for both industry and academia to explore. Now we will explain all the three modules in detail along with the necessary definitions, and design choices made along the way.

4.2 Online AD using diverse RDS fleet data

All of Vista's intelligence depends on identifying accurate anomalous patterns in database performance in real-time. If we accurately detect these bottlenecks with low false alarms, identifying root cause becomes reasonably easier.

Database Load. To prevent the users from having to monitor hundreds of telemetry metrics, Vista derives an interpretable univariate time series metric called *Database Load* (*db_load*) which serves as a proxy of database health. This process is ubiquitous, and appreciated in real-world to reduce the cognitive load of user [7]. A typical database health metric is commonly seen to be derived from average query latency [33, 51], or number of active sessions [5, 13, 29] both of which are highly interpretable. In Vista we define *db_load* as the average number of active sessions per minute. We choose this for 3 key reasons–
1. It gives us a strong baseline: An active session is a connection that has submitted work to the DB engine and is waiting for response. A session is 'active' when it is either running on CPU or waiting for a resource to become available so that it can proceed. For e.g., an active session might wait for a page to be read into the memory, and then consume CPU while it reads data from the page. Thus, when there are more active sessions than the available resource, i.e., number of CPU cores, it means someone is waiting and there is an opportunity for optimization. More concretely, if *db_load* $>$ *num_cpu*, database is under load. This concrete explanation gives us a strong baseline in terms of *num_cpu*, and is simple enough for users to consume and interpret in real-time for decision making.
2. Low computational overhead: Obtaining active sessions is

Table 1: 10 real database performance bottleneck scenarios created by our DBAs to test `Vista`'s RCA ability on both AMS and APG engines. The third column highlights the Dominating Wait Event (DWE) for corresponding problematic SQL query in that problem type.

Engine	Problem Type	DWE	Description
APG	Poor Application Design	`Lock:Tuple`	High number of concurrent sessions trying to acquire conflicting lock for same tuple by running `UPDATE` and `DELETE` statements.
	Poor Transaction Management	`Lock:TransactionId`	Result of long running transactions that hold longer locks blocking other transactions from running or high concurrency.
	Network Congestion	`Client:ClientRead`	Connection is in idle transaction state and is waiting for a client to send more data or issue a command.
	Lock Contention	`LWLock:buffer_content`	High number of concurrent queries updating the same buffer content on tables with a lot of indexes.
	Poor Application Design	`IO:BufFileRead` and `IO:BufFileWrite`	Large number of `ORDER BY` and `GROUP BY` queries consuming the `work_mem area`.
AMS	Storage Latency	`io/aurora_redo_log_flush`	DB doing excessive commits and write operations.
	Workload Spike	`io/table/sql/handler`	Greatly increase the rate of I/O transactions causing a workload spike.
	Poor Application Design	`synch/cond/innodb/row_lock_wait`	One session has locked a row for an update, and another session tries to update the same row.
	Workload Spike	`synch/sxlock/innodb/hash_table_locks`	Pages not found in the buffer pool, thus, must be read from a file.
	Poor Configuration	`synch/mutex/innodb/buf_pool_mutex`	A thread has acquired a lock on the InnoDB buffer pool to access a page in memory.

computationally expensive [29]. To avoid this, we resort to sampling followed by aggregation. The data collection agent wakes up every second, scans the database, and counts the number of sessions that are active. Note that this sampling process is biased towards multi-second long running queries, while the fast queries that are < 1 second are missed. This is a deliberate process as optimizing fast queries isn't our goal. Furthermore, if it is indeed a frequent query, it will likely get sampled in one the runs. Finally, the count is averaged and reported every minute to run AD on it.

3. Highly interpretable. This aggregate *db_load* metric is a counter which can further be factorized down into activity types (for e.g., wait event types) informing the user about what type of load their database is facing, i.e., is it I/O heavy, CPU heavy, and so on. This piece of information becomes valuable later when surfacing useful recommendations.

Forecasting model. With the simple definition of database 'load' discussed above, it may seem like training a deep neural network to detect anomalies is an overkill. But note that real-world workloads are highly noisy and periodic (see Figure 1). Thus, simple rules like $db\_load > N \times num\_cpu$ (N=1,2,3, etc) may lead to 100-1000s of detections per day which is unacceptable. Hence, we need a system that can infer the changing workload pattern, discount periodic spikes in *db_load* metric, and pick out the outliers.

To achieve this, `Vista`'s AD module is built using a neural network based forecasting model which is trained in a self-supervised fashion without needing any external labels. Forecasting based AD is well studied in literature and has shown to work extremely well on variable length time series data [14, 45, 46]. Motivated by effectiveness of MQ-RNN [50], we build a forecasting network based on the RNN Seq-2-Seq model [47]. The recurrent nature of RNNs induces a causal relationship, i.e., forecast for next time step depends only on history, which is key for building a streaming service where future values aren't available to look during online inference. One important distinction from classical forecasting models is that `Vista` AD model's final goal is to produce anomaly segments which we refer to as *Anomaly Events*, instead of mere next step raw forecast. An Anomaly Event is a tuple of form $\{start\_time, end\_time, severity, ongoing\_flag\}$ representing the signature of a detected anomalous incident. The $start\_time$ and end_time are timestamp objects denoting the start and end time of an anomaly, $severity \in$ {`LOW, MEDIUM, HIGH`} is a measure of criticalness of the detected issue, and $ongoing\_flag$ is a boolean value informing whether the anomaly is ongoing (i.e., 1) or has ended (i.e., 0). Later in this section we will discuss how we compute each of these attributes of a given anomaly event.

Since it is technically easier to forecast point estimates rather than long continuous time segments, we modify the forecasting problem as follows: Given a time series of the form $x_{t-i}, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_{t+j}$ we aim to predict the output $\mathcal{F}(x_{t+1}, \ldots, x_{t+j})$ from $x_{t-i}, \ldots, x_t$ where $\mathcal{F}$ is a representation of future values. When $\mathcal{F}$ is an identity function, the problem reduces to a classical multi-step forecasting problem, e.g., DeepAR [44]. In `Vista` we represent $\mathcal{F}$ with the statistical mean of future values, i.e, –

$$\mathcal{F}(x_{t+1}, \ldots, x_{t+j}) = \bar{x}_{t+1:t+j} \tag{1}$$

where $\bar{x}_{t+1:t+j}$ denotes the mean of sequence $[x_{t+1}, \ldots, x_{t+j}]$. Our proposed RNN model for forecasting of mean statistic is similar to Seq-2-Seq model [16, 47] which consists of an encoder and a decoder module. Unlike other forecasting methods [41, 44] who typically aim to predict a sequence, our model predicts only a single variable as shown in Eq.1. This technical modification gives us two benefits– (a) it is a simpler problem to solve; (b) predicting a mean estimate of next few steps, instead of raw value at the next step helps counter the correlation between values at adjacent time points and provides robustness by preventing the forecaster from collapsing into a trivial persistence model [21]. In terms of building blocks of our recurrent model, we chose Gated Recurent Units (GRUs) in the encoder and a Multilayer Perceptron (MLP) in the decoder as shown in Figure 5.

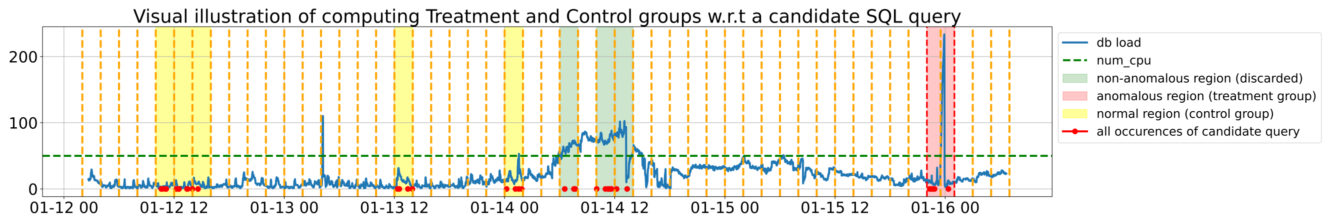


Figure 6: The colored regions indicate all the occurrences of candidate query q_i. The red region indicates the occurrences during the 'anomaly', and forms the *treatment* group, while the yellow regions, which form the *control* group, indicate the query occurrences during the 'normal' times, i.e, when database was healthy ($db\_load < num\_cpu$). Query occurrences in the green regions are ignored by both treatment and control as these regions are neither anomalous, nor deemed healthy.

Skip Connections. Skip connections have been shown as an effective approach to improve modeling dependencies in sequential data, e.g., in CNNs [37] and RNNs [15]. The prior approaches apply skip connections on hidden units while in Vista we apply skip connections directly on the raw inputs. Figure 5 illustrates the modification in the revised network. Instead of feeding a single value x_t from the raw time series, we expand the number of channels at each timestamp t to obtain $\tilde{x}_t$. Subsequently, the corresponding input becomes a vector, for all timestamps. Instead of simply using the past l timestamps to generate $\tilde{x}_t = [x_t, \ldots, x_{t-l+1}]$ which acts as a 'lag-trick' [50] and induces periodic information in input, we introduce a *Periodicity Alignment* feature to handle long range periodic dependencies arising due to daily or weekly periodic jobs run on the database.

Periodicity Alignment. Periodicity alignment is a simple yet efficient approach to enhance long-range temporal dependency modeling. It works by directly appending historical data in the current timestamp, and hence avoids building a more complex model. More specifically, the input to the GRU unit at timestamp t, i.e, $\tilde{x}_t$ is represented as –

$$\tilde{x}_t = [x_t, \vec{x}_{t-\delta}, \vec{x}_{t-2\delta}, \ldots, \vec{x}_{t-c\delta}] \tag{2}$$

where $\delta \in \mathbb{N}^+$ is the length of a period. Note that δ is a hyperparameter which can be tuned based on the fleet data or domain knowledge. The $c \in \mathbb{N}^+$ determines the number of periods to look back. We define $\vec{x}_{t-k\delta}$ as –

$$\vec{x}_{t-k\delta} = [x_{t-k\delta}, x_{t-k\delta+\eta}, \ldots, x_{t-k\delta+s\eta}] \; \forall k = 1, 2, \ldots c \tag{3}$$

Here $\eta \in \mathbb{N}^+$ is the step size to shift right, and $s \in \mathbb{N}^+$ is the number of shifts.

Note that the setup in Eq.(3) via right shift enables spikes in history to appear in one of the channels of data just preceding a future spike, and thus provides essential information for a more accurate prediction. The forecasting model is trained in a self-supervised manner on standardized *db_load* data using a quantile-based loss function [50] and a stochastic gradient descent optimizer ADAM [24]. We define the quantile loss as follows–

$$\mathcal{L}_\tau(\bar{x}, \hat{x}) = \max[\tau(\bar{x}, \hat{x}), (1-\tau)(\hat{x} - \bar{x})] \tag{4}$$

where $\bar{x}$ is the ground truth, $\hat{x}$ the prediction, and $\tau \in (0, 1)$ corresponds to the quantile. Compared to typically used mean squared error (MSE) loss, quantile loss is more robust to outliers and can obtain interval estimation [25]. In production we set $\tau = 0.9$, i.e p90 as the quantile for computing loss, but it is a design choice which should be varied based on the sensitivity of downstream task.

AD via Forecasting. With a trained forecasting model, the next step is to employ it for anomaly detection. Given that Vista's AD is a streaming algorithm, it should commit a decision at the current time based on historical data only. For every timestamp t, Vista's AD is expected to return only one of the following 4 decisions, which is further illustrated in the flowchart in Figure 4.

- **D1**: Start a new anomaly.
- **D2**: Do nothing.
- **D3**: Continue the current ongoing anomaly.
- **D4**: Close the current ongoing anomaly.

In order to reduce the detection latency in production, we minimize the number of calls made to underlying forecasting model. More specifically, we invoke the forecasting model to run a forward pass only when we need to determine whether to start a new anomaly, i.e, D1 above. In order to start an anomaly, the forecasting model's output needs to meet two conditions that we define below in Eq.(5) using these two quantities– (a) Quantile loss $\mathcal{L}_\tau(\bar{x}, \hat{x})$ as shown in Eq.(4); (b) Truth-over-Prediction ratio $\mathcal{R}(\bar{x}, \hat{x}) = \frac{\bar{x}}{\hat{x}}$.

$$\begin{aligned} \mathcal{L}(\bar{x}, \hat{x}) &\geq \theta \\ \mathcal{R}(\bar{x}, \hat{x}) &\geq \epsilon \end{aligned} \tag{5}$$

Where the thresholds θ and ϵ are set empirically. Vista starts an anomaly only if both these conditions are met. Intuitively, the first condition is straightforward: start an

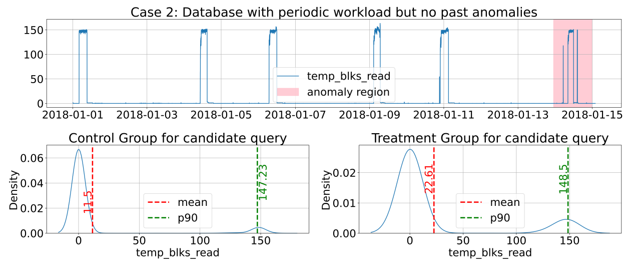


Figure 7: *Using mean to compare long-tail distributions leads to False Positive (FP):* The *temp_blks_read* statistic of candidate query shows a regular periodic behavior as compared to history, but when we compare the *mean* estimate (red) during the anomaly time, to its *mean* w.r.t its history, it doubled indicating a significant change in no. of temporary blocks read by the query and incorrectly concluding it as to be a root cause. However, a *quantile* statistic (green) is more robust in such a case.

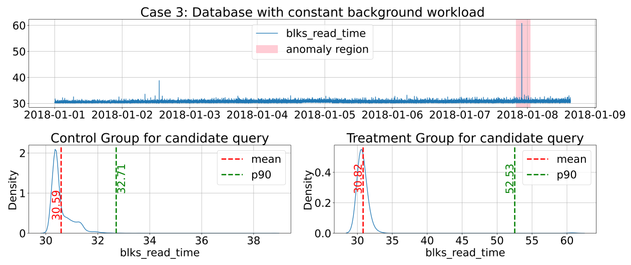


Figure 8: *Using mean to compare long-tail distributions leads to False Negative (FN):* The *blks_read_time* statistic of candidate query shows a thin tall spike during an observed anomaly duration. A *mean* estimate (red) cannot capture this rare event, while a *quantile* (green) does, and flags the candidate query as root cause due to an increase in time spent reading data file blocks w.r.t its historical behavior.

anomaly when the true *db_load* deviates from the predicted *db_load*. However, relying only on the first condition leads to miss-detections (False Negatives) where the quantile loss is impacted by the $\{0, 1\}$ scale of the standardized input data. When both $\bar{x}$ and $\hat{x}$ are small, their quantile loss becomes smaller. To handle this, we use their ratio, which is scale invariant, unlike the quantile loss based on the difference. Once an anomaly is declared, we set its $ongoing\_flag$ to *true* and monitor its progress to decide when to terminate. Since the neural network is only used to check whether to start an anomaly, we instead decide the termination of an anomaly by simply comparing *db_load* against the *num_cpu* metric. More specifically, we close an ongoing anomaly when its *db_load* value is consistently less than *num_cpu* for κ minutes, which is a tunable hyperparameter. Both these processes give us the start and end time of an Anomaly Event (*start_time*, *end_time*). An ongoing anomaly is stored in a persistent storage system (e.g., DynamoDB table [2]) and retrieved for every AD run. Similarly, the *db_load* and *num_cpu* metrics are cached as `model_state` in an S3 bucket. The last key attribute of an Anomaly Event is its severity which quantifies the impact of that anomaly on the database.

Severity. We categorize the severity of an anomaly in 3 levels: {`LOW, MEDIUM, HIGH`}. For an anomaly x starting at t_0 and ending at t_1 we compute a severity score $\mathcal{S}(x)$ as follows–

$$\mathcal{S}(x) = \frac{\bar{x}_{t_0:t_1}}{num\_cpu_{t_0:t_1}} \tag{6}$$

Thus, score is the `mean` of *db_load* between t_0 and t_1, normalized by corresponding *num_cpu*. An implicit requirement of the scoring function is that it should be strictly increasing, i.e., an anomaly with a larger value of *db_load* must have a larger score. We now need score thresholds $\{\omega_1, \omega_2\}$ which can be used to convert the raw scores into the 3 pre-defined *severity* labels as follows–

- $0 \leq S(x) < \omega_1$: then *severity* = `LOW`
- $\omega_1 \leq S(x) < \omega_2$: then *severity* = `MEDIUM`
- $\omega_2 \leq S(x)$: then *severity* = `HIGH`

We compute these thresholds empirically with help from domain experts where we first run `Vista`'s AD module on the RDS fleet for a period of 2 weeks to obtain $\sim 25K$ detections and then use domain experts to filter out 1000 most common anomalies from the fleet. This carefully sampled set serves as a proxy for the larger RDS fleet of 100K instances. Experts then further study these anomalies to segregate them into 3 categories {`LOW, MEDIUM, HIGH`}. We then use a simple distribution matching scheme to infer the severity thresholds from it. More specifically, if the percentage of `LOW`, `MEDIUM` and `HIGH` anomalies in the expert labeled sample set are $q_1\%, q_2\%$ and $q_3\%$ then we compute $\omega_1, \omega_2, \omega_3$ as –

$$\begin{aligned} \mathbb{P}[S(X) < \omega_1] &= q_1/100 \\ \mathbb{P}[\omega_1 \leq S(X) < \omega_1] &= q_2/100 \\ \mathbb{P}[\omega_2 \leq S(X)] &= q_3/100 \end{aligned} \tag{7}$$

Goal here is to end up with thresholds, which, if used to label the severity of anomalies in the sample set, generate the same distribution as the experts. We acknowledge that this is an approximation given the lack of labeled data, but works reasonably well in practice.

Why does a simple static rule suffice to close an anomaly? We use a sophisticated neural net model to start an anomaly, whereas it is closed using a simple static rule of $db\_load > num\_cpu$ consistently for κ minutes. This cuts the number of calls we make to forecaster in half and improves latency. The reason this simple rule is sufficient is because of the way we define database health, i.e., a database is healthy when $db\_load <= num\_cpu$. So if it continues to remain so for $\kappa > 1$ minutes, we can safely assume the database is healthy and close the anomaly without requiring the forecaster to predict it. The reason we can't do the same thing to start an anomaly is because real databases, specially dev databases, are known to run heavy workloads where $db\_load > num\_cpu$ regularly. In such cases, Vista needs to identify and discount these periodic spikes to prevent false alarms for which we need sophisticated models.

Does it work with any RDS engine? Yes. The AD model is DB engine agnostic, i.e., it can be used by customers running their database on any engine (for e.g., Aurora PostgreSQL [10], or Aurora MySQL [9], Oracle [11], etc).

What about an instance with zero history? New customers are key when a service is launched, but unfortunately ML models need history to make accurate predictions. Thus, to ensure a low false positive rate even for customers with no history, we apply a key guardrail: we adopt a *staged* detection process, i.e., we do not surface any detection made by Vista to customer for at-least first ψ_1 minutes, followed by surfacing only HIGH severity detections for next ψ_2 minutes. The intuition is that first stage is warmup phase-1 where the predictions are technically valid but unreliable and could lead to false alarms. Second stage is a warmup phase-2 period where we are confident in surfacing only the most severe anomalies, where $db\_load >> num\_cpu$. The danger of a false alarm is less due to- (a) model has seen more data and hence can make more confident predictions; (b) a severely high spike can potentially be interesting to customer from an optimization perspective, even if it is not statistically anomalous.

What about the case when there's a data loss/corruption, or database crashes? We have two guardrails to handle such scenarios– (a) Since neural-nets are known to generalize well, we exploit this behavior when training the forecaster where we randomly crop small patches of data from input sequence and force the model to predict correct estimates from incomplete data. This form of added regularization helps us handle cases where missing data is small (e.g., few minutes to an hour); (b) For cases where the database crashes, or the loss is for several hours to days, we simply assume the recovered database to be a new instance with zero history and fallback to our staged detection process described in previous question.

There seem to be a lot of magic hyperparameters that need to be set in order for Vista *to work?* In order to deal with a diverse fleet of databases with varying periodicities, and having necessary guardrails to prevent false alarms, we need a list of tunable parameters that can be set externally. We provide a complete list of hyperparameters that need to be set/tuned, some by domain knowledge, some by mere grid search over a wide search space, and some purely driven by design choice for the downstream task. There's no "right" value for these parameters, but nevertheless, we share the values that worked best for us, and the rationale behind choosing them in Table 5.

4.3 Root Cause Analysis with long tail data distribution

Vista's RCA module is based on how expert DBAs debug a performance issue, i.e., by asking *"What changed?"*. This question can be studied across three main categories– (a) User-driven: This refers to analyzing SQL queries run by the user, and pin-pointing the ones that behaved differently during the anomaly as compared to their behavior during normal times (e.g., some query started reading more rows than usual); (b) System-driven: This refers to sub-optimal configurations that lead to performance bottlenecks (e.g., database is under-provisioned, out-of-memory events, etc); (c) External: This refers to issues like Large Scale Events (LSEs) that cause the system to crash. We look at the first category and defer the rest two for the following reasons: (a) System-driven RCA is non-trivial by looking at just telemtery data, for e.g., if the database is under provisioned on CPUs, one can not infer that by just monitoring the *num_cpu* metric which is mostly static throughout. This requires sophisticated models with an understanding of how databases work [48, 52], which is out of scope for this discussion as Vista relies solely on telemetry data for RCA, which is by design, to respect user privacy; (b) Performance issues due to external events like LSEs are highly rare given the stability of production environments, and how robust cloud service providers are.

For user-driven category, we look at individual SQL queries that ran on database during the anomaly detected by Vista, and compare them with a robust baseline to pin point the deviant ones. However, in production there are millions of queries that run every minute so we need a two-stage approach– Step-1: Reduce the search space by filtering out candidate queries for RCA; Step-2: Use robust estimators to compute appropriate baselines for those queries and pinpoint the ones that changed. This type of two-staged approach has shown to work pretty well [29].

4.3.1 Candidate Query Selection.

RDS engines collect a large number of statistics[†] per query, every second, which makes it infeasible to store, and monitor them constantly over time without impacting the overall performance of database. Hence, Vista only collects and analyzes queries when an anomaly is detected. But even during an anomaly event,

[†]List of query statistics: Aurora PostgreSQL, Aurora MySQL

which could last from few seconds to several hours, there can be a few thousands to millions of queries that got executed. Analyzing every single one of them in real-time is still computationally non-trivial. Hence, `Vista` resorts to a filtering mechanism where it flags out only the *"dominating"* or *"candidate"* queries during the anomaly event.

How do you define a "dominating" query? In the process of execution, a query interacts with multiple database and OS sub-systems, such as query plan repository, database buffer cache, IO or networking layers, spending time in each step. As the query goes through all these stages, its current state is continuously being emitted by the database in the form of a *session state* (that we sometimes refer to as wait event). By aggregating these states over time, it becomes possible to estimate where the query is spending most of its time during execution. Such aggregation can also be done on a database level, to determine what majority of database queries are waiting for. `Vista` extracts this information and aggregates it per query during the anomaly event, ranks the queries w.r.t this in descending order, filter out the top-K queries, and refers them as *"dominating"* SQL queries or *"candidate"* queries.

Are SQL queries the true root cause of performance degradation? Not necessarily. Note that, `Vista` relies only on telemetry data (e.g., query statistics) which limits its capability to infer the true root cause (e.g., what the end users are actually doing, e.g., executing more queries, changing schema, upgrading instance class type, etc). However, those unobserved actions impact the query statistics which `Vista` logs and monitors. This is what we refer to as RCA in `Vista`, i.e., surfacing queries that changed behavior during the *db_load* anomaly, so users can dive-deep and identify the true root cause themselves.

So what assumptions are you making in this process? We make three key assumptions: (a) If a performance issue does not manifest itself in *db_load* as an anomaly, `Vista` has no means to detect that issue. It is by choice, as we believe if *db_load* is low, database is healthy, and we do not need to analyze anything; (b) If *db_load* is anomalous, `Vista` assumes something is definitely wrong with the database, and will try to highlight queries that changed behavior during the anomaly. This could be misleading in cases where those queries are not necessarily the root-cause but instead are just impacted by the issue (e.g., query execution time peaked up as the user downgraded the database instance); (c) Invariant characteristics (e.g., fixed parameters or hardware specs) are not monitored and hence excluded from the analysis. These are known limitations, and we defer them to future work as improvements for `Vista`.

4.3.2 *Robust Quantile Detectors for RCA.*

We study the problem of RCA under the Potential Outcomes Framework (POF) [43] known for its intuitive yet rigorous mathematical foundation. It also avoids learning complex causal models from observational data which is non-trivial specially in case of databases where thousands of different queries run on the database every second.

Preliminaries. We first define the key terminologies in POF:
Unit: This is a proxy for an individual/person in a typical POF experiment. We define a *unit* as an occurrence of the candidate query q_i.
Treatment: In POF, a treatment W_i is a binary action performed on the units belonging to the *Treatment Group.* Goal of POF is to then identify if the treatment was effective, i.e., was *treatment* truly the root cause of an *outcome* observed on the treatment group as compared to a baseline, i.e., *control group*? In our context, we consider this binary treatment to be the true unobserved underlying root cause that led to the *db_load* anomaly. Interestingly, we do not need to know the true treatment (a.k.a the root cause) to study its impact using SQL queries. The binary treatment manifests itself in form of a *db_load* anomaly, which we will use to define treatment and control groups below.
Treatment Group: For each candidate query we define a separate treatment group. This group is made up of all the occurrences of query q_i within the anomaly region, denoted by red in Figure 6.
Control Group: For each candidate query we define a separate control group. This group is made up of all the occurrences of query q_i within the normal region in the history, i.e., prior to the observed anomaly. In Figure 6 the five yellow regions indicate the normal region where query q_i was observed. Note that a healthy/normal region is defined using our standard definition of $db\_load < num\_cpu$. Hence, even though q_i was also observed during the green regions, we do not include those occurrences in control group. Excluding unhealthy, and anomalous regions when computing baseline helps build a robust control group and prevents false negatives as explained with a real-world example in Figure 3.
Outcome: We represent the outcome variable Y_i to be a vector representing the impact on query features of q_i. Thus, $Y_i(W_i = 1)$ denotes outcome vector for query q_i when treated $W_i = 1$, and similarly, $Y_i(W_i = 0)$ represents outcome for control. $Y_i \in \mathbb{R}^d$ where d is number of query statistics.
Treatment Effect: To measure the impact of treatment, we compute a score using the two outcomes. This score is usually the difference in means, called Average Treatment Effect (ATE) defined as follows-

$$\zeta = \frac{1}{m}\sum_i Y_i(W = 1) - \frac{1}{n-m}\sum_i Y_i(W_i = 0) \tag{8}$$

Where n is the total number of units, out of which m receive the treatment. This ATE framework can be used to explain the RCA methods adopted by various prior works

Table 2: Subset of system metrics that Vista runs the same RCA module upon, along with SQL queries to provide additional content of performance degradation.

Engine	CPU	Memory	Connections
APG	os.cpuUtilization.total.avg os.tasks.running.avg os.general.numVCPUs.avg	os.memory.outOfMemoryKillCount.avg	db.User.numbackends.avg db.User.total_auth_attempts.avg db.User.total_auth_failed.avg
AMS	os.cpuUtilization.total.avg os.tasks.running.avg os.general.numVCPUs.avg	os.memory.outOfMemoryKillCount.avg	db.Users.Threads_connected.avg db.Users.Connections.avg db.Users.Aborted_connects.avg

that use difference in mean [51], or T-test [29, 33] to pin point deviating queries or metrics.

POF in Vista: Intuitively, Vista's POF setting aims to answer this question: *"Given that an unknown intervention caused a performance degradation observed in db_load as an anomaly (a.k.a treatment), can you explain the issue by identifying the SQL queries that executed during the anomaly (a.k.a treatment group), and deviated (a.k.a outcome) from their usual behavior (a.k.a control group)?"*

In Vista we make two key modifications to the traditional POF in order to make it more robust. First, to handle the long tail behavior observed in query statistics (see Figures 3, 7, 8), Vista computes Quantile Treatment Effect (QTE) instead of ATE. More specifically, we compare $p90$, i.e., 90th percentile of treatment group with $p100$, i.e., max of control group. Thus, we modify Eq.(8) as–

$$\zeta_{q_i} = \max\Big(0, (p90[Y_i(W=1)] - p100[Y_i(W_i=0)])\Big) \quad (9)$$

Setting a higher quantile threshold for control group is a design choice that makes the QTE stricter and less susceptible to false positives. These quantile values should be chosen carefully based on downstream task. We wrap the QTE with a ReLU operator [35] for numerical stability. Second, we use a non-parametric test, like Permuatation test [36] to reject the null hypothesis H_0, which says, the anomaly manifested in *db_load* has no effect on dominating queries rendering them futile for root cause analysis. We repeat the permutation process B times, and use a p-value threshold of α to reject/accept the null hypothesis. We compute the p-value as follows–

$$\text{p-value} = \frac{1}{B}\sum_{b=1}^{B}\mathbb{I}((\zeta_{q_i})_b > \zeta_{q_i}) \quad (10)$$

Where $\mathbb{I}$ is the indicator function. Since, Y_i is a d dimensional vector, we run this analysis for each of d statistics for all K candidate queries, and rank the queries in descending order of their number of statistics that successfully rejected the null hypothesis. For e.g., if for candidate queries q_1 and q_2 we found 3 and 7 statistics rejected their null respectively, we will rank q_2 higher than q_1 denoting q_2 is more important when investigating the corresponding anomaly. Furthermore, our experienced in-house DBAs hand-picked a small subset of key system metrics that have proven to be useful when supplemented with the knowledge of deviant queries for doing RCA. Vista repeats the same RCA process as described above on these system metrics, and combines this information along with identified queries to send to the recommendation module. The complete list of system metrics for both Aurora PostgreSQL (APG), and Aurora MySQL (AMS) engine that Vista uses is provide in Table 2.

Table 3: Comparison of different AD approaches on a test set of $\sim 4.5M$ data points.

Method	Eval type	Precision	Recall	F1
STL	Event	0.759	0.694	0.725
	Segment	0.722	0.666	0.693
	Weighted	0.952	0.818	0.881
SR-CNN	Event	0.561	0.353	0.433
	Segment	0.524	0.312	0.394
	Weighted	0.648	0.495	0.513
Vista	Event	**0.831**	**0.753**	**0.791**
	Segment	**0.774**	**0.725**	**0.747**
	Weighted	**0.962**	**0.924**	**0.943**

4.4 Human-in-the-loop Recommendation Module

The final component of Vista is its recommendation module which we discuss briefly for completeness. We keep this module relatively light-weight and simple as its goal is to supplement Vista's detection and RCA results with right RDS documentation and a short blurb for the user to get started. For e.g., for a database running on APG engine, when Vista detects a *db_load* anomaly that is dominated with sessions waiting for `IO:XactSync` and `Lock:typle`, it will supplement the analysis with the corresponding wait event documents that are carefully curated by expert in-house DBAs[†].

5 EXPERIMENTS

In this section, we experimentally assess the usefulness of Vista's findings both qualitatively and quantitatively. More specifically, we evaluate:

[†]Click to see the list of wait event troubleshooting documents for Aurora PostgreSQL.

Table 4: Qualitative evaluation of `Vista` at identifying the root cause SQL queries during different types of real performance issues (see Table 1 for problem type descriptions).

Engine	Problem Scenario	Precision	Recall	F1
APG	Type 1	0.923	0.781	0.846
	Type 2	1.0	0.879	0.935
	Type 3	0.973	0.959	0.965
	Type 4	1.0	0.628	0.764
	Type 5	0.933	0.810	0.867
AMS	Type 1	0.891	0.772	0.827
	Type 2	0.980	0.901	0.938
	Type 3	0.954	0.977	0.965
	Type 4	1.0	0.723	0.839
	Type 5	0.966	0.834	0.895

1. AD accuracy (Section 5.1): How accurately can `Vista` detect anomalies in real-time on unseen real-world customer databases?
2. AD run-time (Section 5.2): How much time does `Vista` take to return detection results for each sample data point in a streaming manner?
3. Detection latency (Section 5.3): How quickly does `Vista` recognize an ongoing performance degradation?
4. RCA accuracy (Section 5.4): How accurately can `Vista` identify the deviant SQL queries for RCA?
5. Real-world case studies (Section 5.5): How useful is `Vista` when deployed end-to-end in real-world?

What was `Vista` *trained on?* `Vista` was trained on a collection of $\sim 100K$ customer workloads (i.e, *db_load* metric) sampled randomly from the RDS fleet comprising of both APG and AMS databases, for a period of 90 days. Since *db_load* is collected at a per-minute frequency, the total number of training data points used were $\sim 12.96B$. The workloads demonstrated quasi-periodic behavior across multiple periods (for e.g., hourly, daily, or weekly) as shown in Figure 1. As `Vista` was trained in a self-supervised manner, we did not need any labels during the training process. All the hyperparameters used in `Vista` are stated in Table 5 along with the rationale behind choosing those values. The training was performed on a single `p3.8xlarge` instance, using the ADAM [24] optimizer, with the quantile loss stated in Eq.(4), and a learning rate of 10^{-3}. The forecasting model used was a 3-GRU layered encoder and decoder with 100 hidden units each. The total number of parameters were 151K.

5.1 AD accuracy

Our expert DBAs hand-labeled an internal data set comprising of real-world workloads, sampled randomly from 150 anonymized real customer databases, over a period of 3 weeks (i.e., $\sim 4.5M$ data points) to test `Vista` on. The distribution of test set was 3.25% anomalous points, and rest non-anomalous. We acknowledge that this test set depends on a sampling process which even though carried out meticulously by domain experts, is susceptible to incomplete representation of all types of workload present in the fleet. For benchmarks we pick two readily used light-weight AD approaches in industry– (a) STL [17, 26, 34, 49]; and (b) Fourier based SR-CNN [42]. Both these algorithms are unsupervised and do not require labels for training like `Vista`'s AD model. To ensure consistency, we trained/tuned the baselines on the same unlabeled training set which `Vista` was trained on.

Evaluation Metrics. We use 3 different types of evaluation metrics: (a) Event-based; (b) Segment-based; (c) Weighted Precision, Recall and F-1 scores. Event and Segment metrics are based on the amount of overlap defined using `IoU` (Intersection-Over-Union) b/w predicted and true anomalous segments, with an `IoU` threshold of > 0.1 and > 0.9 respectively to determine a true positive detection [53]. Weighted metrics weigh each prediction and ground truth segment with their severity score that gives an estimate of how good the AD algorithm is for more severe anomalies (eg: HIGH severity). This is crucial in real-world because high severity anomalies are prone to causing severe bottlenecks leading to major impact on business and the revenue as compared to short-lived low severity anomalies.

Performance Results. Table 3 shows the performance scores obtained by `Vista` on the internal test set of $\sim 4.5M$ data points. The test was run in a streaming manner, every minute, and the final scores are obtained by averaging 5 test runs. We can see that on database telemtery data, `Vista` AD has 10% better precision, 8% better recall and 9% better F1 score as compared to the second best method.

5.2 AD run-time

Faster run-time is a key requirement for deploying `Vista` in production. We compute a sample-by-sample run-time which is the amount of time taken to run inference on a single new data point in the streaming manner. As compared to the STL-based method and SR-CNN which obtain a sample-by-sample run-time of 0.15 and 0.09 seconds respectively, `Vista` clocks a run-time of 0.024 seconds. These scores represent the 95th percentile (p95) of the run-time distribution computed over the test set.

5.3 Detection latency

Another key production metric is the detection latency which is defined as the amount of delay between the start time of anomaly, and the time when it was surfaced to customer by `Vista`. Ideally we want detection latency to be 0 minutes, i.e real-time detection. On the test-set that contains ~ 3500 anomalous segments, `Vista` obtains a p95 detection latency of 17.47 minutes which is near real-time given our prediction window is 15 minutes (Eq. 1).

Table 5: List of free parameters in Vista along with their description, reference, values used, and the rationale behind it.

Parameter	Description	Reference	Value	Rationale
j	Number of data points in input	Eq.(1)	15	Design choice
δ	Length of a period	Eq.(2)	1440	Grid search on the fleet
c	Number of periods to look back in history	Eq.(3)	3	Grid search on the fleet
η	Step size for right shift	Eq.(3)	2	Grid search on the fleet
s	Number of shifts	Eq.(3)	10	Grid search on the fleet
τ	Quantile value used for loss function	Eq.(4)	0.9	Design choice
θ	Threshold for starting an anomaly	Eq.(5)	0.5	Grid search on the fleet
η	Threshold for starting an anomaly	Eq.(5)	6.5	Grid search on the fleet
ω_1	Threshold for severity score	Eq.(7)	4	Grid search on labeled subset
ω_2	Threshold for severity score	Eq.(7)	8	Grid search on labeled subset
q_1	% of `LOW` anomalies	Eq.(7)	60%	Expert labeling of sampled subset
q_2	% of `MEDIUM` anomalies	Eq.(7)	31%	Expert labeling of sampled subset
q_3	% of `HIGH` anomalies	Eq.(7)	9%	Expert labeling of sampled subset
κ	Minutes to wait before closing an anomaly	Sec.(4.2)	15	Design choice
ψ_1	Warmup (phase-1) time in minutes	Sec.(4.2)	15	Grid search on the fleet
ψ_2	Warmup (phase-2) time in minutes	Sec.(4.2)	180	Grid search on the fleet
K	Number of candidate queries to analyze for RCA	Sec.(4.3)	5	Design choice
$\mathcal{H}$	Length of look back history to compute control group (in minutes)	Sec.(4.3)	1440	Design choice
d	Number of query statistics	Sec.(4.3)	15	Domain Knowledge
α	p-value threshold for permutation test	Sec.(4.3)	0.05	Design choice
B	Number of repetitions in permutation test	Eq.(10)	100	Grid search on the fleet

5.4 RCA accuracy

To test the Vista's accuracy in detecting the root cause SQL query, our expert DBAs create a set of 10 realistic problem types spread across both APG and AMS engine type. Each problem type is a workload with a set of $\sim$ 100-500 SQL queries, amongst which there are 1-5 root cause SQL queries which Vista needs to highlight. Table 1 explains each problem type, what the root cause SQL queries mainly wait for, and a description of the workload. For each problem type, we compute Vista's precision, recall and F1 score. Table 4 shows the scores achieved by Vista. Note, that Vista achieves a higher precision as compared to the recall for almost all problem types. This is due to the higher quantile threshold used for the control group computing the QTE (See Eq.(10)). This forces Vista to have a high tolerance for false alarms, but also leads to missed detections observed in the recall scores.

5.5 Real-world case studies

In this section we share real-world experiences of DBAs interacting with Vista in their day-to-day life to debug performance issues.

5.5.1 Case Study 1: Lock Contention. In this scenario DBA investigates a *db_load* anomaly where 40-100 database sessions are waiting for database to respond on a database with 16 CPU cores. Firstly, the DBA looks at the wait event distribution of the *db_load* anomaly detected by Vista to see what are sessions mainly waiting for. Vista shows that spike is dominated by a single wait event `Lock:tuple` (97% of the entire *db_load* spike). The `Lock:tuple` wait event represents a case where multiple sessions are completing for access to the same database records. Furthermore, Vista identifies two SQL queries that deviated from their normal behavior and had an increase in `execution_rate` and `logical block reads` which correlated to the *db_load* spike. DBA infers that it is a locking issue by looking at the wait event distribution of *db_load* spike, and an increase in the query execution time potentially indicates that there are *popular records* that these queries are continuously competing for. Finally, Vista provides the `lock` related troubleshooting document using which the DBA can get started on implementing fixes and best practices.

5.5.2 Case Study 2: CPU Exhaustion. Vista notifies the DBA about a *db_load* anomaly where 450 sessions are waiting on a database with 2 CPU cores and mainly dominated by `CPU` wait event. Vista also surfaces a query that shows abnormal behavior and contributes 95% to the anomaly in terms of wait events. Furthermore, Vista also highlights severe increase in key system level metrics like `cpuUtilization.total.avg` and `outOfMemoryKillCount.avg` indicating exhaustion of resources. Noting that root cause query's execution rate and logical reads increased very slightly, DBA deduces that its a scenario where heavy sorting (which uses both CPU and memory) is, likely the true root cause of this issue. While Vista does not (yet) have metrics that track CPU and Memory usage per query, it still guides the DBA in right direction by combining the power of system metrics with per-query statistics.

5.5.3 Case Study 3: I/O Issues. Vista notifies the DBA about a *db_load* spike dominated by wait events `IO:DataFileRead` and `IPC:BufferIO` contributing over 87% to the detected anomaly. Vista also highlights the potential root-cause query seen to be waiting for I/O. DBA digs deeper on the query using its `query_id` and finds that it is a range scan on primary key. Reading about the two wait events from the troubleshooting docs provided by Vista, DBA figures that the query is both reading from disk (into database cache) as well as competing for cache access. Furthermore, Vista highlights an abnormal increase in the execution rate and logical reads of the query. Based on the analysis, DBA deduces two insights: (a) Increased execution rate has likely caused increased disk reads by individual queries; (b) Database buffer cache is likely too small to hold all the data in database working set (requiring cache pages to be constantly evicted). DBA arrived at these insights in a couple of minutes, which usually takes several minutes if not hours to discover.

6 A HITCHHIKER'S GUIDE TO DATABASE MONITORING DEPLOYMENTS AT SCALE

Despite careful design and concerns about possible deployment issues, we still faced many challenges. In the following section we try to outline what we have learned from them, by describing many of the problems we faced and how we solved them.

6.1 Data Management

6.1.1 Handling missing values. In production, encountering gaps in streaming data is common due to various reasons (for e.g., data collection agent failure, customer shuts down the database, its a new customer with no history, etc). The AD module should be robust to such gaps and strike a balance between the two scenarios– (a) raising false alarms, i.e, flagging a periodic job as anomaly due to gaps in input history, and (b) being overly conservative, i.e, not detecting any spike due to missing values in the anomalous segment. Vista has put guardrails in place to handle these: (a) When the data gaps are longer (e.g., several hours to days) due to a database crash, shutdown, or in case of a new customer with zero history, Vista adopts a staggered approach where it doesn't surface any detection for first ψ_1 minutes, followed by only surfacing HIGH severity detection for the next ψ_2 minutes. This limits false alarms arising due to lack of input data; (b) Vista regularizes its training process to handle shorter data gaps (e.g., few minutes to an hour) by randomly cropping segments of data out during training to create data gaps and force the neural-net to learn interpolation via generalization.

6.2 Model Learning

6.2.1 Robust predictions. While sequence models like RNNs and LSTMs are among the most popular choices for modeling time-series data for next-step prediction, they suffer from a peculiar problem called *lag-effect* or the *persistence problem* [21] when predicting the value at next time step $\hat{x}_{t+1}$ given the history, i.e $x_{t-k}, \ldots, x_t$. The problem can be described as $\hat{x}_{t+1} \approx x_t$, i.e, the predicted value by the model is same or similar to the value at the preceding timestep t. The model thus learns to simply predict x_t at $t+1$ leading to incorrect forecasts. In order to prevent this, Vista predicts the statistical mean of values at next j timesteps ($j = 15$) instead of predicting the value at next step (Eq.1). This acts as a regularizer by adding noise during training and prevents the model from collapsing into predicting preceding value at each step.

6.3 Model Verification

6.3.1 Human-in-the-loop. We performed several round of DBA reviews to qualitative assess the RCA and recommendations generated by Vista. During these weekly reviews we randomly sampled 100 databases distributed equally across engine-types, and configuration-types from the fleet and ran Vista to obtain detections, RCA and recommendations. These detections where then randomly assigned to a group of experienced DBAs to obtain feedback. This time-consuming human-in-the-loop evaluation helped us fine-tune Vista's analysis module by setting right thresholds, identifying the important system metrics to add, and writing actionable troubleshooting docs to make Vista overall useful.

6.3.2 Choosing right performance metrics. During internal reviews we learned that having a highly precise service (low false alarms) with bounded recall, is more useful for customers than a noisier service with the highest F1 score. Furthermore, production-ready service required establishing a trade-off between high precision, bounded recall and other components like detection latency, memory footprint of input, model size, etc. Tuning model parameters to obtain the best F1 score is not the right strategy when deploying a service in production.

6.4 Model Deployment

6.4.1 Cost considerations. Cost is a key metric to optimize when launching a production service, specially at AWS scale. We optimized cost by cutting the number of invocation of Vista's AD module by 50%, by using it only to decide when to open an anomaly. Choosing right health metric that's simple for customers to understand with a interpretable baseline made the other decisions (e.g., continuing, or closing an anomaly) achievable by simple rules. Furthermore, during months long internal trial runs on the entire fleet, we observed that running the service at 3-minute intervals on any given database led to the same number of detections as running it every minute, which further reduced the cost by $\sim 66.67\%$.

6.4.2 Robust deployments. Failures come in many form like software bugs, infra failures, or operational mistakes which can bring the service down making it unreliable for businesses. To prevent this, we adopted a cell-based architecture

[4] to deploy Vista across RDS fleet that helps isolate failures to a single cell instead of affecting the entire region. Secondly, it also enables horizontal scaling while maintaining maximally sized components that can easily be performance tested. Finally, as the fleet grows, a new cell can easily be added to handle demand increase.

7 DISCUSSION & USER FEEDBACK

Post launch we interacted with customers to understand what are their pain-points when using Vista and how can we address them. A key request from the customers was to have a way to control the sensitivity of Vista's detection mechanism, both in terms of length of anomaly, and its severity. This is key because some databases tend to be noisier than others, which can also change with time, so a knob to make detection stricter would help them reduce false alarms. In order to address this we adopt a 2-phase solution. Firstly, we allowed customers to simply configure alerts in terms of severities, i.e., they could archive all the LOW, MEDIUM anomalies, and get alerts only for HIGH severity anomalies. However, this still doesn't meet the needs of those customers who wanted more control. To address that, we are working with customers to get more feedback and building a feature where we expose a sensitivity slider with 5 levels (0, 25, 50, 75, 100) that can help them control the sensitivity of the detector. Under-the-hood this slider will map to a fixed set of key hyperparameters (e.g., minimum length of anomaly, threshold for severity score, vCPU level) , with the highest sensitivity generating the least number of alerts and lowest sensitivity the most number of alerts. These key parameters are selected based on several customer interactions by our domain experts, and the values are set using grid-search run over the large RDS fleet.

8 CONCLUSION

Vista is an ML-based performance troubleshooting framework serving in production for > 2 years, making thousands of critical detections preventing $> 10K$ false alarms every day for RDS customers. Future work will focus on adding more system metrics and per-query features in RCA, moving to a multivariate AD system by factorizing the aggregate *db_load* metric into corresponding wait events for more granular insights, and building an anomaly deduping mechanism to reduce cognitive overload, specially for customers that monitor a fleet of 100-1000s of databases.

REFERENCES

[1] [n. d.]. Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.
[2] [n. d.]. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.
[3] [n. d.]. Azure Monitor. https://learn.microsoft.com/en-us/azure/azure-monitor/overview.
[4] [n. d.]. Cell-based Architecture. https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/what-is-a-cell-based-architecture.html.
[5] [n. d.]. Oracle. Average Active Session. [Online]. https://docs.oracle.com/en-us/iaas/performance-hub/doc/view-average-active-sessions-data.html.
[6] [n. d.]. Oracle Enterprise Manager. https://www.oracle.com/enterprise-manager/.
[7] [n. d.]. OtterTune Health Score. https://docs.ottertune.com/support/kb/articles/BWzAvY9E/health-score.
[8] [n. d.]. Performance Insights. https://aws.amazon.com/rds/performance-insights/.
[9] 2024. Amazon Aurora MySQL. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.AuroraMySQL.html.
[10] 2024. Amazon Aurora PostgreSQL. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.AuroraPostgreSQL.html.
[11] 2024. RDS for Oracle. https://aws.amazon.com/rds/oracle/.
[12] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. ACM| Proceedings of the 2022 International Conference on Management of Data.
[13] Wei Cao, Yusong Gao, Bingchen Lin, Xiaojie Feng, Yu Xie, Xiao Lou, and Peng Wang. 2018. Tcprt: Instrument and diagnostic analysis system for service quality of cloud databases at massive scale in real-time. In *Proceedings of the 2018 International Conference on Management of Data*. 615–627.
[14] Chris U Carmona, François-Xavier Aubet, Valentin Flunkert, and Jan Gasthaus. 2021. Neural contextual anomaly detection for time series. *arXiv preprint arXiv:2107.07702* (2021).
[15] Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark A Hasegawa-Johnson, and Thomas S Huang. 2017. Dilated recurrent neural networks. *Advances in neural information processing systems* 30 (2017).
[16] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).
[17] Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning. 1990. STL: A seasonal-trend decomposition. *Journal of official statistics* 6, 1 (1990), 3–73.
[18] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.
[19] Jingkun Gao, Xiaomin Song, Qingsong Wen, Pichao Wang, Liang Sun, and Huan Xu. 2020. Robusttad: Robust time series anomaly detection via decomposition and convolutional neural networks. *arXiv preprint arXiv:2002.09545* (2020).
[20] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, and Amit Levi. 2020. Sentinel: universal analysis and insight for data systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2720–2733.
[21] Espen Haugsdal, Erlend Aune, and Massimiliano Ruocco. 2023. Persistence initialization: A novel adaptation of the transformer architecture for time series forecasting. *Applied Intelligence* (2023), 1–16.
[22] Shiyue Huang, Ziwei Wang, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. 2023. DBPA: A Benchmark for Transactional Database Performance Anomalies. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
[23] Vimalkumar Jeyakumar, Omid Madani, Ali Parandeh, Ashutosh Kulshreshtha, Weifei Zeng, and Navindra Yadav. 2019. ExplainIt!–A declarative root-cause analysis engine for time series data. In *Proceedings of the 2019 International Conference on Management of Data*. 333–348.

[24] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[25] Roger Koenker. 2005. *Quantile regression*. Vol. 38. Cambridge university press.

[26] Sooyeon Lee and Huy Kang Kim. 2019. Adsas: Comprehensive real-time anomaly detection system. In *Information Security Applications: 19th International Conference, WISA 2018, Jeju Island, Korea, August 23–25, 2018, Revised Selected Papers 19*. Springer, 29–41.

[27] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3028–3042.

[28] Ping Liu, Yu Chen, Xiaohui Nie, Jing Zhu, Shenglin Zhang, Kaixin Sui, Ming Zhang, and Dan Pei. 2019. Fluxrank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 35–46.

[29] Xiaoze Liu, Zheng Yin, Chao Zhao, Congcong Ge, Lu Chen, Yunjun Gao, Dimeng Li, Ziting Wang, Gaozhong Liang, Jian Tan, et al. 2022. PinSQL: Pinpoint Root Cause SQLs to Resolve Performance Issues in Cloud Databases. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2549–2561.

[30] Xianglin Lu, Zhe Xie, Zeyan Li, Mingjie Li, Xiaohui Nie, Nengwen Zhao, Qingyang Yu, Shenglin Zhang, Kaixin Sui, Lin Zhu, et al. 2022. Generic and Robust Performance Diagnosis via Causal Inference for OLTP Database Systems. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 655–664.

[31] 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 *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[32] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: decomposed behavior modeling for self-driving database management systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1248–1261.

[33] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, et al. 2020. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1176–1189.

[34] Abhinav Mishra, Ram Sriharsha, and Sichen Zhong. 2021. OnlineSTL: scaling time series decomposition by 100x. *arXiv preprint arXiv:2107.09110* (2021).

[35] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[36] Anders Odén and Hans Wedel. 1975. Arguments for Fisher's permutation test. *The Annals of Statistics* (1975), 518–520.

[37] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[38] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[39] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make your database system dream of electric sheep: towards self-driving operation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3211–3221.

[40] Anthony N Pettitt. 1979. A non-parametric approach to the change-point problem. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 28, 2 (1979), 126–135.

[41] Syama Sundar Rangapuram, Matthias W Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. 2018. Deep state space models for time series forecasting. *Advances in neural information processing systems* 31 (2018).

[42] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. 2019. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3009–3017.

[43] Donald B Rubin. 2005. Causal inference using potential outcomes: Design, modeling, decisions. *J. Amer. Statist. Assoc.* 100, 469 (2005), 322–331.

[44] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. 2020. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting* 36, 3 (2020), 1181–1191.

[45] Dominique T Shipmon, Jason M Gurevitch, Paolo M Piselli, and Stephen T Edwards. 2017. Time series anomaly detection; detection of anomalous drops with limited features and sparse examples in noisy highly periodic data. *arXiv preprint arXiv:1708.03665* (2017).

[46] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. 2017. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1067–1075.

[47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

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

[49] Qingsong Wen, Jingkun Gao, Xiaomin Song, Liang Sun, Huan Xu, and Shenghuo Zhu. 2019. RobustSTL: A robust seasonal-trend decomposition algorithm for long time series. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 5409–5416.

[50] Ruofeng Wen, Kari Torkkola, Balakrishnan Narayanaswamy, and Dhruv Madeka. 2017. A multi-horizon quantile recurrent forecaster. *arXiv preprint arXiv:1711.11053* (2017).

[51] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 international conference on management of data*. 1599–1614.

[52] Ji Zhang, Ke Zhou, Guoliang Li, Yu Liu, Ming Xie, Bin Cheng, and Jiashu Xing. 2021. CDBTune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *The VLDB Journal* 30, 6 (2021), 959–987.

[53] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. 2019. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems* 30, 11 (2019), 3212–3232.