RobustScaler: QoS-Aware Autoscaling for Complex Workloads

Huajie Qian, Qingsong Wen, Liang Sun, Jing Gu, Qiulin Niu, Zhimin Tang

DAMO Academy, Alibaba Group
Bellevue, WA, USA

{h.qian, qingsong.wen, liang.sun, zibai.gj, qiulin.nql, zhimin.tang}@alibaba-inc.com

Abstract—Autoscaling is a critical component for efficient resource utilization with satisfactory quality of service (QoS) in cloud computing. This paper investigates proactive autoscaling for widely-used scaling-per-query applications where scaling is required for each query, such as container registry and function-as-a-service (FaaS). In these scenarios, the workload often exhibits high uncertainty with complex temporal patterns like periodicity, noises and outliers. Conservative strategies that scale out unnecessarily many instances lead to high resource costs whereas aggressive strategies may result in poor QoS. We present RobustScaler to achieve superior trade-off between cost and QoS. Specifically, we design a novel autoscaling framework based on non-homogeneous Poisson processes (NHPP) modeling and stochastically constrained optimization. Furthermore, we develop a specialized alternating direction method of multipliers (ADMM) to efficiently train the NHPP model, and rigorously prove the QoS guarantees delivered by our optimization-based proactive strategies. Extensive experiments show that RobustScaler outperforms common baseline autoscaling strategies in various real-world traces, with large margins for complex workload patterns.

Index Terms—autoscaling, point process, stochastic constraint, time series, AIOps

I. INTRODUCTION

Autoscaling has been a fundamental tool in elastic cloud services that dynamically adds (scales out) or deletes (scales in) computing resources, such as instances, CPU and memory, to closely match the ever-changing computing demand. Real-world workload can change drastically over time, and reactive autoscaling that conducts scaling actions only after the workload changes often leads to underprovision during increasing traffic and in turn degradation of quality of service (QoS). Therefore, proactive autoscaling is becoming increasingly popular in that it scales out beforehand when a demand increase is expected so that resource shortfalls can be avoided and satisfactory QoS is consistently maintained.

In this paper, we consider proactive autoscaling in the scaling-per-query scenario of on-demand cloud service: When a query comes in, it will be served by one of the available idle instances (warm start) or a new instance will be initiated for that query (cold start); meanwhile, each instance is terminated after processing one query and is not reused/shared by other queries. The scaling-per-query scenario is common in cloud computing: One example is the container registry for storing, managing, and securing custom container images; another example can be found in Continuous Integration (CI) and Continuous Delivery (CD) to avoid user’s account/data being hijacked/spooofed. Note that scaling-per-query is in contrast to other cloud services where an instance keeps running and processing new queries until it is terminated for other reasons (e.g., scaled in when traffic goes down). A purely reactive scaling mechanism in the scaling-per-query scenario would simply create an instance for every new query and terminate the instance after processing the query. The main drawback of this mechanism is the cold-start delay, i.e., the startup time of an instance is not negligible compared to the actual processing time of the query, making the total response time unnecessarily long. One way to reduce the cold-start delay is to maintain a pool of running instances in anticipation of future queries so that processing can start immediately upon arrival of a query, however, this can incur a considerable overhead cost since the instances may stay idle without processing any query, especially when the pool size is large, and it is not clear what pool size is needed to achieve a certain QoS.

In practice, proactive autoscaling also encounters several challenges. One challenge lies in capturing the complex and variable periodic patterns in the workload. The workloads of database and cloud computing often exhibit notable periodic patterns. By identifying and utilizing the periodicity, we can perform effective autoscaling of resources in these scenarios to save a significant amount of resources. However, real-world workload may come with lots of missing data, anomalies, large noises that, on one hand, obscure many structural patterns such as periodicity that are crucial for prediction, and on the other hand requires strong robustness from the workload prediction algorithm. We leverage robust decomposition to extract periodic patterns from the queries-per-second (QPS) time series for query arrival modeling and forecasting, even in the presence of a considerable amount of noises, missing data and anomalies. Another challenge concerns the control of the trade-off between resource saving and QoS improvement. As more resources are allocated, the QoS gets improved, and vice versa. For scaling-per-query, the more instances are maintained and the earlier they are created, the less cold starts will occur. Proper control of this trade-off becomes more challenging for scaling-per-query due to its per-query scaling dynamics. For example, the lifecycle (hence the cost) of an instance also depends on the arrival and the execution time of the query, and whether cold start occurs is
related to the temporal order between the arrival time of the query and the time the instance finishes startup which is not known beforehand when creating the instance. Therefore, even evaluating the cost and QoS (e.g., cold-start delay) requires knowledge of the query arrival dynamics. Moreover, instances created before will be consumed by incoming queries and therefore the number of available instances changes after every query arrival, making scaling-per-query a sequential decision making problem. All these necessitate the modeling of the query arrival process for which we develop a non-homogeneous Poisson process (NHPP) framework with novel periodicity regularization that can flexibly approximate variable periodicity patterns. Moreover, we formally investigate the trade-off through the lens of stochastically constrained optimization \cite{20} from which scaling decisions that respect either QoS constraints or cost constraints can be efficiently computed.

The main goal of this study is a proactive autoscaling framework that can properly create instances before the queries arrive to guarantee a certain QoS with a low overhead cost. To the best of our knowledge, no previous work has investigated proactive autoscaling to reduce cold-start delay in the scaling-per-query scenario. To this end, we design a novel autoscaling framework called RobustScaler which can generate robust scaling decisions that optimally balance the trade-off between cost and QoS and are robust to noise, missing data and anomalies. In summary, our contributions are: 1) develop the first NHPP framework for demand modeling in autoscaling to capture both periodicity and stochasticity of query arrivals, along with an efficient alternating direction method of multipliers (ADMM) training algorithm; 2) propose a stochastically constrained optimization formulation to compute Pareto-optimal scaling decisions in terms of cost and QoS; 3) design a sequential scaling scheme based on the stochastically constrained optimization that enjoys provable probabilistic QoS guarantees; 4) conduct an extensive set of experiments that demonstrate the superiority of RobustScaler compared to heuristic strategies.

The rest of the paper is organized as follows. Section II reviews the related literature. Section III introduces the setting and challenges for scaling-per-query. Section IV describes the proposed autoscaling framework. Section V then presents our NHPP arrival model with periodicity regularization, whereas Section VI details the QoS-cost trade-off in scaling decision making, the constrained optimization formulations, and the consequent sequential scaling schemes. Section VII presents experimental results including a comparison of the proposal with two heuristic autoscaling methods. Section VIII concludes the paper.

II. RELATED WORK

Autoscaling is an active research area in various types of cloud systems, such as database \cite{21, 22}, microservices \cite{23–25}, stream processing \cite{26–28}, and web applications \cite{29, 30}. Recent surveys include \cite{2, 31–34}. These autoscaling systems have been successfully applied in their respective types of systems, but cannot handle the scaling-per-query scenario considered in this paper where the key difficulty is to mitigate cold start for future queries. Similar cold start issues also appear in serverless FaaS platforms, and are typically mitigated by either maintaining a pool of pre-warmed instances \cite{35, 36} or reusing warm instances after they finish current function invocations \cite{37}. However, unlike the serverless environment, instances are not reused in our case and hence the scaling dynamics are different.

Based on the scaling timing, autoscaling methods can be categorised into proactive and reactive ones \cite{2}. Note that the full potential of proactive autoscaling can be utilized when the workload has predictable and periodic/cyclic patterns. Fortunately, many real-world workloads exhibit such patterns \cite{12–17}. However, in practice we often face highly noisy and variable periodic patterns with missing data and outliers \cite{3, 18}, which calls for robust workload prediction. For example, P-Store \cite{3} is a predictive autoscaling scheme for database systems under workload with a diurnal pattern, but it does not automatically capture other periodicity patterns. Meanwhile, P-store does not characterize uncertainties in workload forecasting. Periodic workload patterns are also utilized in the Turbine of Facebook \cite{26} for stream processing systems, where it forecasts future load patterns in a heuristic way without carefully considering pattern variations or outliers. In contrast, we leverage robust decomposition \cite{18, 19} of time series in workload forecasting to deal with complex variable periodic patterns under challenging noise and outliers.

To properly characterize uncertain query arrivals in the workload, a conventional modeling tool is Poisson processes \cite{7, 38} that is analytically tractable with elegant statistical properties and accurate enough when the traffic consists of many independent users with weak temporal dependency \cite{39}. Otherwise, Markovian arrival processes \cite{40} and hierarchical bundling models \cite{41} have been developed to capture bursts and temporal dependency. See \cite{16} for a comprehensive survey. However, in these approaches periodicity patterns are either not considered or assumed not to vary over time, whereas we adopt NHPP with a novel periodicity regularization to flexibly approximate variable periodicity patterns.

Existing autoscalers generate scaling decisions based on control theory \cite{42, 43}, reinforcement learning \cite{45, 46}, queuing theory \cite{47, 48}, and rule-based methods \cite{49}. Most of these methods either make scaling decisions based on a mean demand estimate without considering uncertainty, or handle uncertainty in a heuristic way. For example, model predictive control is adopted for predictive autoscaling in \cite{42} which utilizes ARMA model for workload forecasting and look-ahead controller for resource allocation without considering uncertainty. The autoscaling scheme RobustT2Scale \cite{43} integrates a fuzzy controller with an online learning mechanism that can cope with certain uncertainties but is not general enough to handle variable periodic patterns. Instead, we directly incorporate workload uncertainty into scaling decisions via stochastic constraints that are expressive of QoS requirements to derive robust scaling decisions.
instance stays idle for a time span of $\xi$, is said to be "hit" and can be processed right away, but the instance gets ready (e.g., the first query in Figure 1), the query is formalized in Algorithm 1.

If the arrival time is earlier than the ready time (e.g., the second query), the query is not hit and incurs an unnecessary cost. If the arrival time is later than the ready time (e.g., the fourth query), it will be immediately created to process the query, and the originally scheduled creation at time $x_i$ is canceled. Finally, once the instance finishes processing the query, it gets deleted immediately. These dynamics are illustrated before in Figure [1] and balancing their trade-off serves as a fundamental challenge in making scaling decisions.

### Algorithm 1 Scaling dynamics

Given the instance creation times $x_1, x_2, \ldots$

1. for $i = 1, 2, \ldots$ - th query do
2.   if $x_i + \tau_i \leq \xi_i$ then
3.     the query is processed immediately
4.   else if $x_i \leq \xi_i < x_i + \tau_i$ then
5.     the query waits until $x_i + \tau_i$ and then processing starts
6.   else
7.     an instance is created to process the query
8. end if
9. end for

### III. THE AUTOSCALING PROBLEM

Formally, let $\Delta t$ be a fixed time step, and $\{Q_t : t = 1, \ldots, |T|\}$ be a sequence of historical query counts from an application deployment within each time interval of length $\Delta t$, that is, $\frac{Q}{\Delta t}$ is the QPS. Suppose the current time is 0, and let $0 < \xi_1 < \xi_2 < \cdots < \xi_i < \cdots$ be an ascending sequence of random arrival times of the queries, where $\xi_i$ is the time of arrival of the $i$-th upcoming query, let $0 < x_1 < x_2 < \cdots < x_i < \cdots$ be an ascending sequence of deterministic instance creation times (the $i$-th instance to be used to process the $i$-th query). Let $s_1, s_2, \ldots, s_i, \ldots$ be a sequence of i.i.d. random processing times of the queries, and $\tau_1, \tau_2, \ldots, \tau_i, \ldots$ be a sequence of i.i.d. random pending/startup times of the instances. Therefore the $i$-th instance shall get ready for processing at time $x_i + \tau_i$. Let $\mu_s := E[s_i], \mu_\tau := E[\tau_i]$ be the expected values of each $s_i, \tau_i$ respectively. To illustrate the dynamics of the query arrivals, their interplay with scaling actions, and the associated resource cost, Figure [1] shows a sample of the query arrival process. If the arrival time $\xi_i$ of the $i$-th query is later than the time $x_i + \tau_i$ when the $i$-th instance gets ready (e.g., the first query in Figure 1), the query is said to be "hit" and can be processed right away, but the instance stays idle for a time span of $\xi_i - (x_i + \tau_i)$ which incurs an unnecessary cost. If the arrival time is earlier than the ready time (e.g., the second query), the query is not hit and needs to wait until the instance gets ready to start processing. Otherwise if the $i$-th instance is not even created before $\xi_i$ (e.g., the fourth query), it will be immediately created to handle the query and the originally scheduled creation at time $x_i$ is canceled. Finally, once the instance finishes processing the query, it gets deleted immediately. These dynamics are formalized in Algorithm 1.

Our framework aims to recommend the scaling action sequence $0 < x_1 < x_2 < \cdots < x_i < \cdots$ based on historical data so that

- (QoS) the response time of a query, i.e., the time between its arrival and the completion of its processing, is short,
- (cost) the idling time of an instance, i.e., the period between the time it gets fully started and the time it starts to process a query, is short.

Note that these two metrics compete against each other as illustrated before in Figure [1] and balancing their trade-off serves as a fundamental challenge in making scaling decisions.

### IV. FRAMEWORK OF ROBUSTSCALER

In this section we introduce the framework of the proposed RobustScalor method. Specifically, it consists of four main components, including periodicity detection, historical query arrival modeling, query arrival prediction, and scaling plan, as illustrated in Figure 2. In the following we introduce each component in detail.

The first periodicity detection module takes in historical query data and discovers any present cyclic patterns by leveraging robust periodicity detection [15]. Depending on the size of fluctuation of the traffic and the time resolution of the data, periodicity patterns can be obscured by the inherent randomness of the traffic and thus can not be easily detected. Therefore, to reduce random effects and reveal potentially hidden periodicity patterns, we perform a time aggregation by averaging the QPS series in each time window of a fixed size and then perform periodicity detection on the aggregated series.

The second historical query arrival modeling module is the key modeling step that learns the stochasticity of the query arrival process and makes it possible 1) to make fine-grained query-wise scaling decisions and 2) to make robust scaling decisions that are guaranteed to attain a certain QoS level under traffic uncertainty. In particular, a regularized NHPP with periodicity penalty is used, a more detailed exposition for which shall be given in Section [V]. Note that this step on its own is a workload modeling technique that is generally applicable to other settings, although the focus of this work is on scaling-per-query.

The third query arrival prediction module extrapolates the trained point process model to infer the dynamics of the
incoming traffic in the near future. In the case of NHPP, the estimated historical intensity is extrapolated to predict future intensity of the arrival process.

The last scaling plan module utilizes the predicted traffic dynamics to create scaling plans, i.e., a list of scaling in/out actions to be performed in the near future, through a stochastically constrained optimization formulation that characterizes a trade-off between resource cost and QoS. User-specified QoS or cost levels can be directly fed into the formulation to obtain scaling decisions that satisfy the stipulated QoS/cost requirements. Section [VI] discusses in detail the stochastically constrained formulation and our sequential scaling scheme.

V. QUERY ARRIVAL MODELING

This section explains the details of our second module that models the query traffic as a point process [7]. Specifically, we consider modeling query arrivals as an NHPP. Suppose that within each time step \( \Delta t \) the Poisson intensity is a constant \( \lambda_t := \exp(r_t) \), where an exponential transform is adopted to ensure the positiveness of the intensity, so that the query count during the \( t \)-th time interval has a Poisson distribution with rate \( \exp(r_t) \Delta t \). Then the likelihood of observing \( Q_t \) queries within the \( t \)-th time interval is \( \operatorname{exp}(r_t) \Delta t) \exp(-\exp(r_t) \Delta t) / Q_t! \) and the negative log-likelihood of query count series \( \{Q_t: t = 1, \ldots, T\} \) is \( \ell_k(r) \propto -Q^T r + \Delta t \cdot 1^T e^r + \beta_1 \| y \|_1 + \beta_2 / 2 \| z \|_2^2 \)

where the matrix \( D^2 \) is the second order differential matrix to capture the smoothness between every set of three consecutive points [50, 51], \( \beta_1 \geq 0 \) is the smoothness regularization parameter, \( L \) is the period length obtained from periodicity detection, the matrix \( D_L = [I_L - 0 \cdot (T - L) \times L]^T \) is the \( L \)-step forward differential matrix to capture the smoothness across period length \( L \), and \( \beta_2 \) is the periodicity regularization parameter. Additionally, the time step \( \Delta t \) for the QPS series also affects the accuracy of the arrival model. Ideally, \( \Delta t \) shall be chosen by cross validation, but in practice, if the query arrival rate is not expected to vary drastically over time, using a time step in minutes yields reasonably accurate arrival models.

It is not efficient to directly solve the optimization problem [1]. Therefore, we design a quadratically approximated ADMM algorithm to efficiently solve it. Introducing auxiliary variables \( y = D^2 r \) and \( z = D_L r \), we have the augmented Lagrangian for [1]

\[
L_{\rho}(r, y, z; y, z) = -Q^T r + \Delta t \cdot 1^T e^r + \beta_1 \| y \|_1 + \beta_2 / 2 \| z \|_2^2 + \nu^T (y - D^2 r) + \nu^T (z - D_L r) + \rho / 2 \| y - D^2 r \|_2^2 + \rho / 2 \| z - D_L r \|_2^2
\]

where \( \rho > 0 \) is the penalty parameter, and \( \nu^T, \nu \) are dual variables. Note that solutions to the subproblems with respect to \( y \) and \( z \) can be computed from the proximal operators of \( \| \cdot \|_1 \) and \( \| \cdot \|_2^2 \) respectively in closed form. The subproblem with respect to \( r \), however, does not admit a closed-form solution due to the exponential term in the loss. To ensure fast ADMM iterations, we leverage linearized ADMM to solve a quadratically approximated subproblem which can be efficiently solved. Specifically, given the \( k \)-th iterate \( (r_k, y_k, z_k, \nu_y, \nu_z, \nu_{y,k}, \nu_{z,k}) \), we consider the second order Talyor approximation to the exponential term in the augmented Lagrangian

\[
\hat{L}_{\rho}(r, y, z; y, z) = L_{\rho}(r, y, z; y, z) - \Delta t \cdot 1^T e^r + \Delta t \cdot 1^T e^{r_k} + \Delta t \cdot (e^{r_k})^T (r - r_k) + \Delta t / 2 \| r - r_k \|_2^2 \]

where \( \operatorname{diag}(e^{r_k}) \) denotes the diagonal matrix with \( e^{r_k} \) on the diagonal, and update the iterate \( r_{k+1} = \arg\min_r \hat{L}_{\rho}(r, y, z; y, z) \) which reduces to solving a sparse linear system. There are other ways to approximate the subproblem (e.g., by linearizing the proximal penalty only or both the loss and the proximal penalty), but this particular way performs empirically the best for our problem.

The complete ADMM scheme is summarized in Algorithm 2 where the “SoftThreshold” in line 3 is the soft thresholding operator defined as \( \operatorname{SoftThreshold}(x, c) := \operatorname{sign}(x) \max(|x| - c, 0) \).

We remark on the computational complexity of the ADMM iteration. The bottleneck of each iteration in Algorithm 2 lies in solving the linear system \( A_k^{-1} B_k \). The size of the matrix \( A_k \) is the same as the length of QPS time series, i.e., \( T \), and note that \( A_k \) is sparse banded with a bandwidth of order \( O(L) \) (\( L \) being the length of period) and a total of \( O(T) \) non-zero entries due to the structure of the differential matrices
$D^2$ and $D_L$. Therefore, the general computational cost in solving $A^{-1}_k B_k$ is $O(TL^2)$ (see Section 2.4 in [20]). In our experiments below, $T$ is in tens of thousands and the linear system can be solved fairly fast.

VI. SCALING DECISION

This section elaborates on the last module of our framework, where the final scaling plan is derived from the predicted traffic dynamics. We first introduce the QoS and cost metrics and their trade-off, and then present the optimization formulations ([2], [4], and [6]) and the final scaling scheme Algorithm [4].

A. The QoS-Cost Trade-Off

We measure QoS by two criteria:

Response Time (RT): The RT of each query is the time span between its arrival and the completion of its processing as

\[
\begin{align*}
\text{RT}_i &= \begin{cases} 
\tau_i + s_i, & \text{if } \xi_i < x_i \text{ (instance not created)} \\
x_i + \tau_i + s_i - \xi_i, & \text{if } x_i \leq \xi_i \leq x_i + \tau_i \text{ (instance pending)} \\
\xi_i - x_i, & \text{if } x_i + \tau_i < \xi_i \text{ (instance ready)}
\end{cases}
\end{align*}
\]

or equivalently in a more compact form $\text{RT}_i = s_i + (\tau_i - (\xi_i - x_i)_+)_+$, where $(\cdot)_+ := \max(\cdot, 0)$. $s_i$ is the irreducible processing time and $(\tau_i - (\xi_i - x_i)_+)_+$ is the waiting time that can be controlled by choosing $x_i$. We shall use the expected RT $E[\text{RT}_i] = \mu_r + E[(\tau_i - (\xi_i - x_i)_+)_+]$ as the first criterion.

Hitting Probability (HP): The HP of a query is the probability that the corresponding instance is ready for processing upon arrival. Formally, the HP of $i$-th query is $\text{HP}_i = P(\xi_i > x_i + \tau_i)$.

From a cloud user’s perspective, RT seems more tangible than HP, and here HP is considered as an alternative in that it is a good proxy for RT and as a metric is more universal as the range is always $[0, 1]$. In particular, note that both HP and the expected RT are monotone in $x_i$, and when the pending time $\tau_i = \mu_r$ is a constant, it is not hard to see that $E[\text{RT}_i] \leq \mu_s + \mu_r (1 - \text{HP}_i)$. Therefore the expected RT can be controlled below a given threshold by making the HP higher than a corresponding level.

We measure the resource cost of an instance by its lifecycle length, i.e., the time span between its creation and deletion. The total resource cost is the total length of all instances’ lifecycles. To calculate the lifecycle length, note that if the query arrives before the instance becomes ready then the instance starts processing the query immediately after it is ready, hence the lifecycle length in this case is the pending time plus the processing time, i.e., $\tau_i + s_i$; Otherwise the instance stays idle for a certain amount of time until the query arrives, leading to a lifecycle length $\xi_i + s_i - x_i$. Putting both cases together gives the cost of the $i$-th instance as $\text{cost}_i = (\xi_i - x_i - \tau_i)_+ + \tau_i + s_i$ where $\tau_i + s_i$ is the irreducible fixed cost of pending and processing times, and the first term is the idling time. Note that an earlier instance creation time $x_i$ improves the QoS but increases the cost $\text{cost}_i$. Therefore, the key challenge lies in balancing the fundamental trade-off between QoS improvement and cost reduction.

B. Optimization Formulations

We use a constrained optimization to formally study the aforementioned trade-off. Assuming the current time is zero, we consider the planning of the next $K$ instances.

1) QoS-constrained formulation: Using HP as the QoS criterion, we can specify a desired service level $1 - \alpha$, and consider minimizing the expected total cost of the next $K$ instances subject to each query’s HP being above $1 - \alpha$.

\[
\min_{x_1, \ldots , x_K \geq 0} \sum_{i=1}^{K} E[\text{cost}_i], \quad i = 1, \ldots , K
\]

s.t. $P(\xi_i > x_i + \tau_i) \geq 1 - \alpha$

where the fixed cost $\mu_r + \mu_s$ is ignored in the objective for simplicity. The stochastic constraint here is a chance constraint which is notoriously hard to solve because of non-convexity [20]. Fortunately, this problem is separable into $K$ individual subproblems, each having a single decision variable $\min_{x_i \geq 0} E[(\xi_i - x_i - \tau_i)_+]$ s.t. $P(\xi_i > x_i + \tau_i) \geq 1 - \alpha$ which is equivalent to $\max x_i$ s.t. $P(\xi_i > x_i + \tau_i) \geq 1 - \alpha$ by monotonicity whose optimal solution is

\[
x_i^* = \alpha \text{ quantile of } (\xi_i - \tau_i).
\]

Therefore the optimal solution of [2] is $(x_1^*, \ldots , x_K^*)$ with each $x_i^*$ given by [3].

Note that, however, the right hand side of [3] can be negative, due to the pending time $\tau_i$, in which case a $1 - \alpha$ hitting probability is unachievable for the $i$-th query and the problem [2] becomes infeasible. This potential infeasibility suggests the necessity of proactively creating the $i$-th instance in advance in order to achieve the desired QoS level.

Alternatively, replacing the HP with the expected RT as the QoS metric yields an optimization with similar structure. In particular, with a user-specified threshold $d$ for the expected RT, the RT-based formulation

\[
\min_{x_1, \ldots , x_K \geq 0} \sum_{i=1}^{K} E[(\xi_i - x_i - \tau_i)_+], \quad i = 1, \ldots , K
\]

s.t. $\mu_s + E[(\tau_i - (\xi_i - x_i)_+)_+] \leq d$

is also separable and has the optimal solution

\[
x_i^* \text{ such that } E[(\tau_i - (\xi_i - x_i^*)_+)_+] = d - \mu_s.
\]

2) Cost-constrained formulation: If a user prioritizes cost control over QoS, a cost budget $B$ can be specified and we flip the objective and constraint in [4] to solve

\[
\min_{x_1, \ldots , x_K \geq 0} \sum_{i=1}^{K} E[(\xi_i - x_i - \tau_i)_+], \quad i = 1, \ldots , K
\]

s.t. $E[(\xi_i - x_i - \tau_i)_+] + \mu_r + \mu_s \leq B$

where the fixed processing time $\mu_s$ is ignored in the response time objective. Again this problem is separable as for [2] and its optimal solution is

\[
\begin{align*}
\{ x_i^* = 0, & \quad \text{if } E[(\xi_i - x_i)_+] \leq B - \mu_r - \mu_s \\
x_i^* \text{ s.t. } E[(\xi_i - x_i)_+] = B - \mu_r - \mu_s & \quad \text{otherwise}
\end{align*}
\]
Algorithm 3 Sort-and-Search for solving (5)

input: Monte Carlo samples $\tau_i, \xi_i, r = 1, \ldots, R, d, \mu_s$
1: Sort $\{\xi_i\}_{i=1}^R$ and $\{\xi_i - \tau_i\}_{i=1}^R$ into ascending order $\{\xi_i(1)\}, \ldots, \{\xi_i - \tau_i(1)\}$
2: Let $r_1 = 1, r_2 = 2, sl = \frac{1}{R}$, $\hat{E}_l = \hat{E}_r = 0, x_l = x_r = (\xi_i - \tau_i)^{(1)}$
3: while True do
4: if $r_1 > R$ and $r_2 > R$ then
5: $\hat{x}_i = \xi_i(R)$ and break
6: else if $r_1 > R$ or $\xi_i(r_1) \leq (\xi_i - \tau_i(r_2)$ then
7: $sl' = sl - \frac{1}{R}, x_l = x_u, x_u = \xi_i(r_1), r_1 = r_1 + 1$
8: else if $r_2 > R$ or $\xi_i(r_1) > (\xi_i - \tau_i(r_2)$ then
9: $sl' = sl + \frac{1}{R}, x_l = x_u, x_u = (\xi_i - \tau_i)^{(r_2)}, r_2 = r_2 + 1$
10: end if
11: $\hat{E}_l = \hat{E}_r$, and $\hat{E}_p = \hat{E}_p + sl(x_u - x_l)$
12: if $\hat{E}_l \leq d - \mu_s \leq \hat{E}_r$ then
13: $\hat{x}_i = (d - \mu_s - \hat{E}_l)/sl + x_l$ and break
14: end if
15: $sl = sl'$
16: end while
output: $\hat{x}_i$

Solution Method and Complexity: Solving (3), (5) and (7) for the optimal scaling decisions boils down to stochastic root finding problems, which in general can be approximated via Monte Carlo sampling. We provide a sort-and-search algorithm (Algorithm 3) for approximately solving (3), and similar ones can be designed for (5) and (7) as well. Suppose a total of $R$ Monte Carlo samples, $\tau_i, \xi_i, r = 1, \ldots, R$ are drawn to form $\hat{E}(x) := \sum_{r=1}^R (1 - \xi_i - x_+ + \tau_i)/R$ as an approximation of $E[(\tau_i - \xi_i - x_+)]$. Note that $\hat{E}(x)$ is piece-wise linear and monotonic in $x$ and its slope changes only at $\{\xi_i\}_{i=1}^R$ and $\{\xi_i - \tau_i\}_{i=1}^R$, thus the idea is to iterate over these linear pieces from the left to the right until the piece that contains the value $d - \mu_s$ is reached. Specifically, steps 4-10 of Algorithm 3 update the slope at each breakpoint (minus $\frac{1}{R}$ when passing a $\xi_i$, plus $\frac{1}{R}$ when passing a $\xi_i - \tau_i$), step 11 updates $\hat{E}(x)$ at the endpoints $x_l, x_u$ using the slope in $O(1)$ time rather than the direct summation that costs $O(R)$ time. Since the sorting step can be done in $O(R \log R)$ time, and the search step has a linear time complexity, therefore the overall complexity of Algorithm 3 is $O(R \log R)$. The number of actions $K$ to consider usually scales linearly with the QPS of the traffic, hence the practical computational cost is roughly $O(QPS \cdot R \log R)$. The linear complexity in both QPS and sampling size makes our method highly scalable both in theory and practice.

C. Proactive Strategies and QoS Guarantees

As briefly mentioned in the last subsection, proactively scaling out instances is necessary for consistently maintaining a desired level of QoS, and this subsection presents a novel proactive strategy. We will use HP as our QoS criterion.

We explain the main idea of our proactive strategies. A naive strategy based on the solution (3) is to plan the instance creation time at the current time for a batch of $K$ upcoming queries, and then wait until all planned $K$ instances are consumed to do planning for the next batch of $K$ queries. A key issue of this naive strategy however is that we may not be able to have the instances ready in time for the first few queries in each batch, because all previously created instances have been consumed at the time of planning. Therefore, a crucial step is to start planning for the next batch of queries once the number of instances left reaches a certain carefully chosen threshold $\kappa$, rather than after all are used, so that the first few upcoming queries are taken care of by the left instances and only queries that arrive further later need to be included in planning. Algorithm 4 details our proposal.

To explain Algorithm 4 in line 1 the aforementioned threshold $\kappa$ is calculated in (8) by assuming the queries arrive according to a constant intensity $\lambda$ that upper bounds the true $\lambda_i$. According to the way $\kappa$ is computed and the discussion in Section VI-B if the query arrivals follow the constant intensity $\lambda$, for each $k > \kappa$, one is able to get the instance ready in time for the $k$-th upcoming query to achieve a $1 - \alpha$ hitting probability. Since the actual $\lambda_i \leq \lambda$, the $k$-th upcoming query under the actual intensity always arrives later than that under $\lambda$, and thus the $1 - \alpha$ hitting probability must be attainable under the actual intensity too. Lines 2-10 then updates the instance creation plan every $m$ query arrivals. Note that in each round of planning we only need to calculate instance creation times for the $(k + 1)$-th to $(k + m)$-th upcoming queries because the first $k$ upcoming queries already have their instances scheduled in the last round. This way, we always plan instance creation times at least $\kappa + 1$ arrivals ahead, therefore the desired $1 - \alpha$ hitting probability can be achieved for every single arrival.

We provide two propositions below on the hitting probability achieved by Algorithm 4 under different conditions. Proposition 1 shows that the hitting probability is exactly
1 − α when the underlying query process is an NHPP and the intensity is known, serving as an ideal case for Algorithm 1. In practice, however, the intensity needs to be predicted or estimated with a certain error, and in this case Proposition 2 shows that the hitting probability error grows at most linearly in the relative error of the intensity estimate that is used to make scaling decisions, therefore the attained hitting probability quickly approaches 1 − α as the intensity estimate becomes more accurate. They are stated formally as:

**Proposition 1.** If the query arrivals \{\xi_i : i \geq 1\} follow an NHPP with intensity \(\lambda_i, t \geq 0\) and Algorithm 2 is used with \(\lambda_i\) as an input, then \(P(\xi_i \geq x_i^* + \tau_i) = 1 - \alpha\) for every \(i > \kappa\). Moreover, for every \(N > \kappa\) the variance of the hitting ratio

\[
\text{Var}\left(\frac{1}{N-\kappa} \sum_{i=\kappa+1}^{N} 1(\xi_i \geq x_i^* + \tau_i)\right) \leq \frac{2(\kappa+m)(1-\alpha)}{N-\kappa}.
\]

Note that Proposition 1 also shows that the empirical hitting ratio has a quickly diminishing variance as the queries accumulate and hence should be close to the target \(1 - \alpha\).

**Proposition 2.** Suppose that the queries arrive according to an NHPP with unknown intensity \(\lambda_i\), and that Algorithm 2 is used with an estimate \(\lambda_i\) such that \(|\lambda_i - \lambda_i^*| \leq \epsilon \lambda_i^*\) for some small \(\epsilon > 0\), then for each \(i > \kappa\) the hitting probability

\[
P(\xi_i \geq x_i^* + \tau_i) - (1 - \alpha) \leq \frac{\epsilon}{1 - \epsilon} \left(g_{\kappa+m,\alpha} + \mu_\epsilon \sup \lambda_i\right)
\]

where \(g_{\kappa+m,\alpha}\) is the \(\alpha\)-quantile of the Gamma distribution with shape \(\kappa + m\) and scale 1.

We sketch the main ideas in proving the two results. In Proposition 1 the exact \(1 - \alpha\) hitting probability can be established because scaling for each incoming query is planned sufficiently ahead of time as explained before. The variance of the hitting ratio of the first \(N\) queries can be controlled because scaling is planned at most \(\kappa + m\) steps ahead and hence by the independent-increment property of a Poisson process the hitting events of two queries are statistically independent whenever they are at least \(\kappa + m\) apart. This is why the usual reciprocal relationship of variance and sample size \(N\) shows up with the additional factor \(\kappa + m\) in the numerator. Proposition 2 is proved by first representing the hitting probability in terms of Gamma distribution functions through a time rescaling of the NHPP into a homogeneous one, so that the error analysis of hitting probabilities boils down to that of the associated integrated intensities, and then bounding the error of the latter.

The detailed proofs are as follows:

**Proof for Proposition 2.** Denote by \(\mathcal{H}_t\) the arrival history, i.e., a list of times of arrivals, up to time \(t\), and by \(h_i := 1(\xi_i \geq x_i^* + \tau_i)\). For each \(i > \kappa\), let \(c_i\) be the integer such that \(\kappa + cm < i \leq \kappa + (c_i + 1)m\), then the design of Algorithm 1 ensures that \(x_i^*\) is planned when the \(cm\)-th query arrives, therefore by the way \(x_i^*\) is calculated in (7), \(P(\xi_i \geq x_i^* + \tau_i) = E[P(\xi_i \geq x_i^* + \tau_i | \mathcal{H}_{cm})] = E[1 - \alpha] = 1 - \alpha\).

To calculate the variance of the empirical hitting ratio, we first note that, because of the independent-increment property of NHPP and that the horizon of the scaling plan is \(\kappa + m\), \(h_i\) and \(h_j\) are independent whenever \(|i - j| \geq \kappa + m\).

Secondly, by Cauchy-Schwartz inequality, \(\text{Cov}(h_i, h_j) \leq \sqrt{\text{Var}(h_i)\text{Var}(h_j)}\). Therefore, we can write

\[
\text{Var}\left(\sum_{i=\kappa+1}^{N} 1(\xi_i \geq x_i^* + \tau_i)\right) \leq \sum_{i=\kappa+1}^{N} \text{Var}(h_i) + \sum_{i \neq j} \text{Cov}(h_i, h_j)
\]

\[
\leq (N - \kappa)(1 - \alpha) + \sum_{i=\kappa+1}^{N} \sum_{j:0 \leq |j-i| < \kappa + m} \text{Cov}(h_i, h_j)
\]

\[
\leq 2(\kappa + m)(N - \kappa)(1 - \alpha),
\]

and dividing the right hand side by \((N - \kappa)^2\) gives the desired variance bound.
This concludes the error bound in the proposition.

Some practical guidelines for using Algorithm 4 are as follows. First, Proposition 1 suggests a QoS variance that scales with $\kappa$, thus we suggest using estimates of the local intensity at the current time instead of a global upper bound $\hat{\lambda}$ in (8) to obtain a smaller value for $\kappa$ so that the QoS is stabelier. Second, the hitting probability can be calibrated for better accuracy by choosing a set of nominal levels $0 < p_1 < \cdots < p_B < 1$ to run Algorithm 4 on training data so that the resulting actual hitting probabilities $0 < \hat{p}_1 < \cdots < \hat{p}_B < 1$ cover a wide range of $[0, 1]$. This gives a mapping between the nominal and actual hitting probabilities, which can then be used to pick the right nominal level to ensure a desired actual hitting probability.

VII. EXPERIMENTAL RESULTS

We evaluate and compare various proactive autoscaling methods on three real-world datasets to demonstrate the advantages of the proposed RobustScaler. We also evaluate and analyze various aspects, such as scalability, robustness, accuracy, and performance in real environments, of our framework.

A. Algorithms, Datasets, and Metrics

1) Autoscaling Algorithms: We consider two heuristic autoscaling strategies and three variants of our RobustScaler:

- Backup Pool (BP): This strategy constantly maintains a pool of $B$ instances. Upon each query arrival, one of the instances is used to process the query, and then the pool is immediately replenished with a new instance. Using $B = 0$ is equivalent to a purely reactive strategy.

- Adaptive Backup Pool (AdapBP): This heuristic is an adaptive version of Backup Pool where the pool size is regularly adjusted according to the QPS level. Specifically, the average QPS during the most recent ten minutes is used as an estimate of the current arrival rate, and every ten minutes the pool size is reset to be the estimate multiplied by a pre-fixed constant.

- RobustScaler-HP: This is our framework with scaling scheme Algorithm 4 but with two changes: 1) $\kappa$ is chosen time-dependent according to the local intensity at each planning step to improve cost efficiency, and 2) planning frequency is specified by a fixed time interval of $\Delta$ seconds instead of number of queries, meaning that planning is performed every $\Delta$ seconds and in each round instance creation times that lie within the next $\Delta$ interval are all computed.

- RobustScaler-RT and RobustScaler-cost: The same as RobustScaler-HP except that the scaling decisions are computed from the solution (5) and (7), respectively.

Note that many existing autoscaling algorithms rely on system metrics such as CPU and memory utilization, whereas in the scaling-per-query scenario the time of query arrival and its uncertainty are the main factors considered in making scaling decisions, therefore these method cannot handle the scaling-per-query scenario directly, and we do not include them (such as Horizontal Pod Autoscaler (HPA) scheme in Kubernetes [53]) in comparison.

2) Datasets: We test the autoscaling methods on 3 real-world traces. The first trace, “CRS”, is from the container registry service of a top cloud service provider that contains time information of a total of 21059 queries for building container images, including start and end times of query processing, over 4 weeks. The first three weeks of the CRS trace are used as training data and the last week for testing. The second dataset is the Google cluster data 2019 trace [54] that describes workloads running on eight Borg clusters for the month of May 2019. We use the job trace data of “cluster b” and use a subset of 24 hours for the experiments which has 20254 job queries in total, among which the first 18 hours of data are training and the last 6 hours are testing data. The third dataset is the Alibaba Cluster Trace 2018 [55] that contains job information of around 4000 machines over 8 days. We select 5 days of data with a total of 503850 job records, among which the first four days are for training and the last day for testing. We find that the selected parts of Google and Alibaba trace are representative of the original traces in terms of workload patterns and at the same time allow faster experimental runs and discoveries. We replay these real-world traces with various autoscalers to evaluate their performance. Figure 3 plots the QPS series of the three traces after aggregating job/query counts with a time resolution $\Delta t = 60$ seconds. We can see that the CRS trace is quite noisy because of the relatively low traffic but seems to have a weekly pattern, and that Alibaba trace and Google trace both have recurrent spikes but the Alibaba trace has an unexpected burst/spike on the fourth day which brings challenges to modeling and prediction. Therefore, these datasets are challenging enough for differentiating different autoscaling algorithms.

3) Metrics: We use the following evaluation metrics:

- hit_rate: the proportion of queries for which at least one instance is ready upon arrival
- total_cost: the sum of lifecycle lengths in seconds of all instances
- relative_cost: ratio of total_cost versus the cost of the pure reactive BP with $B = 0$
- rt_avg: average response time in seconds of all queries.

B. Experimental Results

1) Comparisons of Autoscaling Methods: A main goal of the experiments is to compare the Pareto efficiency of these autoscalers in handling the QoS-cost trade-off. To this end, for
For each of these autoscalers we vary the parameter that controls this trade-off over a sufficiently wide range and analyze the corresponding QoS and cost metrics. Specifically, in BP the pool size $B$ is varied from 0 to 8 for CRS trace, from 0 to 450 for Alibaba trace, and from 0 to 40 for the Google trace, so that the resulting hitting probability covers a sufficiently wide range in $[0,1]$. In AdapBP the pre-fixed constant is varied from zero to hundreds for each trace. Similarly, in our three RobustScaler variants the corresponding constraint values (the target HP, RT, or cost) are varied. The planning frequency $\Delta$ is set to 1 second for our RobustScalers.

To visualize the comparison, two types of Pareto plots, hit_rate vs relative_cost and rt_avg vs relative_cost, are given for each dataset. On each Pareto plot, each line represents a certain autoscaler and each point on the line is the (hit_rate, relative_cost) or (rt_avg, relative_cost) pair generated by the autoscaler with a particular value of the controlled parameter. Therefore, in the hit_rate vs relative_cost plot, the closer a line stays to the top-left, the more efficient the corresponding autoscaler is in the sense that a higher hitting probability is achieved with the same cost. Similarly, in the rt_avg vs relative_cost plot lines that are closer to the bottom-left corner are better autoscalers.

Figure 4 summarizes all Pareto plots, in which Figures 4(a), 4(c), 4(e) are hit_rate vs relative_cost plots and Figures 4(b), 4(d), 4(f) are rt_avg vs relative_cost plots. These plots clearly shows the QoS-cost trade-off for each autoscaler. E.g., in CRS

(a) hit_rate vs relative_cost on CRS trace (b) rt_avg vs relative_cost on CRS trace

(c) hit_rate vs relative_cost on Alibaba trace (d) rt_avg vs relative_cost on Alibaba trace

(e) hit_rate vs relative_cost on Google trace (f) rt_avg vs relative_cost on Google trace

Fig. 4. Pareto plots of different autoscalers.

(a) variance vs mean of hit_rate (b) variance vs mean of rt_avg

Fig. 5. Variance of QoS on CRS trace.

(a) perturbation size $c = 1$ (b) perturbation size $c = 2$

(c) perturbation size $c = 4$ (d) perturbation size $c = 6$

Fig. 6. Comparison of rt_avg and relative_cost under perturbed data.

(a) perturbation size $c = 1$ (b) perturbation size $c = 2$

(c) perturbation size $c = 4$ (d) perturbation size $c = 6$

Fig. 7. Comparison of hit_rate and relative_cost under perturbed data.
trace, as the pool size of \( BP \) increases from 0 to 8, the relative cost increases to 2.2 whereas the hit_rate increases from 0 to 1 and rt_avg goes down from above 192 to below 180. Similar phenomena persist for our RobustScaler variants as the target hitting probability, response time, or cost varies, as well as for AdapBP.

Compared to the heuristic strategy BP, RobustScaler-HP and RobustScaler-RT are consistently better in that they achieve higher hitting rate or lower average RT than BP under the same cost in all the considered cases. All these gains are tied to our stochastically constrained optimization formulation that explicitly quantifies and optimizes the trade-off. RobustScaler-cost also outperforms BP in most cases, except for some cases with large relative costs on the CRS trace. However, we argue that in these high-cost scenarios the potential marginal gain in hitting probability and response time is already quite small, e.g., the hit_rate is already around 95% and the rt_avg only slightly above 180 (minimum is slightly below 180) when the lines of RobustScaler-cost and BP cross. Therefore from a user’s perspective a relatively low overhead cost might be preferred because of the larger marginal gain in QoS, for which all our RobustScaler variants are more efficient than BP. AdapBP outperforms BP on the CRS and Alibaba traces, which is expected since the pool size is regularly adjusted to match the QPS to avoid wasting instances when queries are rare, however slightly underperforms BP on the Google trace.

Compared to AdapBP, all our RobustScaler variants are superior on the Alibaba and Google traces as they all achieve better response time (Figures 4(d), 4(f)) or hitting probability (Figures 4(c), 4(e)) under the same cost. On the CRS trace (Figures 4(a), 4(b)) the result is mixed: The RobustScaler variants outperform AdapBP when the cost is relatively low, but gradually catch up with or even surpass AdapBP (e.g., RobustScaler-HP and RobustScaler-RT) as the cost increases. Although RobustScalers do not perform the best in low-cost cases, they deliver a much stabler hitting probabilities and response times across time as the plots of QoS variance in Figure 5 show. Specifically, Figure 5(b) is plotted as follows: We first collect the response times of all the queries, and with the queries ordered by their times of arrival we average the response times of every 50 queries, and then calculate the variance of all these averaged response times which is plotted against the mean response time of all queries. Figure 5(a) is constructed via a similar procedure on hitting rate. Therefore, each line there clearly shows the variability of QoS at different levels of modules 1-3 of our model is around 100 seconds with three weeks of data (CRS trace) and no more than 7 seconds with four days of data (Alibaba trace), which is short enough given that the NHPP model only needs to be retrained at a low frequency (e.g., every half an hour), and the runtime of the fourth module to generate scaling decisions is under 5 millisecond for all the three traces which therefore allows for almost instant updates of scaling decisions to accommodate rapidly changing traffic.

Motivated by the unstable QoS of AdapBP, we conduct further experiments to compare AdapBP as RobustScaler as perturbations of growing sizes are introduced into the CRS trace, and the results summarized in Figures 6 and 7 show that AdapBP is more sensitive to data changes and its performance quickly deteriorates to a worse level than RobustScalers as the perturbation size grows. The CRS trace is perturbed as follows: On one hand, starting from the beginning of the trace, for every one hour queries within a five-minute time window are deleted; on the other hand, starting from the sixth minute of the trace, for every one hour \( c \) more times of queries are added to a five-minute time window. AdapBP and RobustScaler-HP are then applied to the perturbed trace with increasing perturbation size \( c = 1, 2, 4, 6 \). Their relative performance in terms of response time and hitting rate are plotted in Figures 4 and 7 respectively, both of which clearly show that as \( c \) increases from 1 to 6 RobustScaler-HP is closing the performance gap at low-cost scenarios and finally becomes superior to AdapBP globally. This demonstrates the weakness of the simple heuristic AdapBP in handling complex workloads and the robustness of our RobustScaler.

Regarding the choice between the three RobustScaler variants, we find that none of them dominates the other two in all scenarios, but we do see that RobustScaler-HP and RobustScaler-RT perform very similarly on all the three traces. Figure 5 also shows that RobustScaler-HP and RobustScaler-RT deliver stabler QoS than RobustScaler-cost, which is related to the imposed constraints on QoS metrics in their mathematical formulations 2 and 4. Therefore, we generally recommend RobustScaler-HP and RobustScaler-RT, whereas RobustScaler-cost is preferred in scenarios with a strict cost budget constraint since RobustScaler-cost places an explicit constraint on the cost (see Section VII-B4 and Figure 10(c) for the accuracy of cost control).

2) Scalability Towards High QPS: We investigate how favorably the computation cost of our framework scales with the system workload. In the three real-world traces with a typical QPS between 0.03 and 6.3, the training time of modules 1-3 of our model is around 100 seconds with three weeks of data (CRS trace) and no more than 7 seconds with four days of data (Alibaba trace), which is short enough given that the NHPP model only needs to be retrained at a low frequency (e.g., every half an hour), and the runtime of the fourth module to generate scaling decisions is under 5 millisecond for all the three traces which therefore allows for almost instant updates of scaling decisions to accommodate rapidly changing traffic.

In order to further test the computational efficiency when facing even higher QPS, we expand the experiment to a simulated trace with a much higher QPS level up to
Specifically, we use intensity function \( \lambda(t) = 1000 \cdot 4^t \cdot \left( \frac{t}{3600} \right)^{40} \left( 1 - \frac{t}{3600} \right)^{40} + 0.001 \) for \( t \in [0, 25200] \) that has an exact period of 3600 seconds to generate query arrival data over a duration of 7 hours. Note that the intensity, or the average QPS, ranges from 0.001 to 10000 which is way beyond that of the three real-world trace. A fixed pod pending time of 13 seconds is used, and an exponentially distributed query processing time with a mean of 20 seconds is used in simulation. The first 6 hours of the generated data are then used as training data and the last 1 hour as test data to evaluate our framework. All the three variants (RobustScaler-HP, RobustScaler-RT, RobustScaler-cost) are tested, where scaling decisions are updated every 5 seconds using Monte Carlo approximation with a fixed sample size 1000. The runtime of each decision update is summarized in Figure 8 and the corresponding QoS metrics are shown in Table I.

The scattered plot for runtime and QPS in Figure 8 clearly demonstrates the high scalability of all the three variants of RobustScaler. In particular, the method can still deliver scaling decision in seconds even when the QPS is in thousands which is far beyond that of the three real-world traces. The plot also shows a linear growth of the runtime relative to QPS, which is in accordance with our theoretical investigations in Section VI-B. In the case of higher QPS, the computation of decisions takes longer and hence the decisions are delayed and may become obsolete by the time they become available. This, however, can be alleviated by extending the planning horizon to compute instance creations for the next “\( \Delta + \text{delay} \)” instead of \( \Delta \) seconds (see the description in Section VII-A1) to compensate for the delay, or by using less Monte Carlo samples in Algorithm 3 to speed up the computation. Overall the computational burden of RobustScalers scales favorably with the workload size. In addition, such a level of scalability does not come at the expense of the accuracy of scaling decisions, since the Monte Carlo approximated decisions with sample size 1000 seem good enough for maintaining a QoS level that is close to the target level set for each variant, as shown in Table I. Note that in Table I the target level, 1 second, for response time by RobustScaler-RT is set with the processing time excluded, i.e., for the quantity \( d - \mu_s \) in (5). Similarly, the target cost level, 2 seconds per instance, set by RobustScaler-cost, is for the average idling time of an instance.

3) Robustness Against Missing Data and Anomalies: We test the robustness of our autoscalers on Alibaba trace that has an unexpected burst/anomalies, and also on the CRS trace by injecting missing data. In particular, we erase the burst in the Alibaba trace to make the pattern more clear and obvious, and for CRS we inject missing data by removing all the queries in one entire day of the fourth week. We then rerun the experiments with the new modified traces. If the QoS and cost metrics generated by the autoscaler before and after the modifications are similar then the autoscaler can be considered robust against anomalies and missing data. Figure 9 summarizes the results for RobustScaler-HP and RobustScaler-cost and it is clear that the resulting QoS and cost metrics are almost identical on the original and modified traces. We also examine the corresponding high-level quantiles (75\%, 95\%, 99\% and 99.9\%) of the response times to better characterize the sensitivity to anomalies and missing data. It turns out that both RobustScaler variants have the exactly same response time quantiles at all considered levels before and after removing the anomalies on Alibaba trace, and the changes of the quantiles on the CRS trace are very tiny as shown in Table II. In addition, the comparison with AdapBP under perturbed QPS data in Figures 6 and 7 also reveals the stable performance of RobustScaler under data perturbations. All these demonstrate the robustness of our autoscalers against missing data and anomalies.

4) Accurate Control of QoS/Cost Levels and Effects of Planning Frequency: Our theory suggests that RobustScaler is guaranteed to achieve a pre-specified hitting probability if the arrival process is approximately Poisson, and here we test the accuracy of RobustScaler in maintaining a pre-specified level of QoS in practice on the CRS trace. Figures 10(a)

| QoS levels | RobustScaler-HP | RobustScaler-RT | RobustScaler-cost |
|------------|----------------|----------------|------------------|
| Target level | 0.9 | 1 | 2 |
| Achieved level | 0.99 | 0.51 | 2.50 |

TABLE I
ACCURACY OF ROBUSTSCALERS WITH MONTE CARLO APPROXIMATION ON SIMULATED DATA

TABLE II
COMPARISON OF RESPONSE TIME QUANTILES BEFORE AND AFTER MISSING DATA INJECTION ON CRS TRACE
In this paper we propose a novel proactive autoscaling scheme RobustScaler for the scaling-per-query cloud computing scenario, which achieves superior trade-off between cost and QoS and is robust to noise, missing data and anomalies. We leverage NHPP with specialized regularization techniques to flexibly capture both periodicity and stochasticity of query arrivals with an efficient ADMM solution. Furthermore, we formulate a stochastically constrained optimization and a sequential scaling scheme with provable probabilistic performance guarantees to achieve better scaling decisions than heuristic strategies. Our extensive experiments demonstrate the

VIII. CONCLUSION

In this paper we propose a novel proactive autoscaling scheme RobustScaler for the scaling-per-query cloud computing scenario, which achieves superior trade-off between cost and QoS and is robust to noise, missing data and anomalies. We leverage NHPP with specialized regularization techniques to flexibly capture both periodicity and stochasticity of query arrivals with an efficient ADMM solution. Furthermore, we formulate a stochastically constrained optimization and a sequential scaling scheme with provable probabilistic performance guarantees to achieve better scaling decisions than heuristic strategies. Our extensive experiments demonstrate the
effectiveness and efficiency of RobustScaler in both simulated and real environments.

References

[1] N. R. Herbst, S. Kounen, and R. Reussner, “Elasticity in cloud computing: What it is, and what it is not,” in 10th international conference on autonomic computing (ICAC ’13), 2013, pp. 23–27.

[2] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-scaling web applications in clouds: A taxonomy and survey,” ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–33, 2018.

[3] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade, “P-store: An elastic database system with predictive provisioning,” in Proceedings of the 2018 International Conference on Management of Data (SIGMOD ’18), 2018, pp. 205–219.

[4] Q. Rebjock, V. Flunkert, T. Januschowski, L. Callot, and J. Castellon, “A simple and effective predictive resource scaling heuristic for large-scale cloud applications,” in VLDB 2021 Workshop on Applied AI for Database Systems and Applications (AIDB ’21), 2020.

[5] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniec, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand et al., “Autopilot: workload autoscaling at google,” in Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys ’20), 2020, pp. 1–16.

[6] D. Tsoumakos, I. Konstantinou, C. Boulmpouka, S. Sioutas, and N. Koziris, “Automated, elastic resource provisioning for NoSQL clusters using TIRAMOLA,” in 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 34–41, 2013.

[7] N. Mahmoudi and H. Khazaei, “Performance modeling of serverless computing platforms,” IEEE Transactions on Cloud Computing, vol. 01, no. 01, pp. 1–15, 2020.

[8] A. Y. Wong, E. G. Chekole, M. Ochoa, and J. Zhou, “Threat modeling and security analysis of containers: A survey,” arXiv preprint arXiv:2111.11475, 2021.

[9] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, “[FaaSNet]: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute,” in 2021 USENIX Annual Technical Conference (USENIX ATC ’21), 2021, pp. 443–457.

[10] A. Anwar, M. Mohamed, V. Tarasov, M. Lintley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig et al., “Improving docker registry design based on production workload analysis,” in 16th USENIX Conference on File and Storage Technologies (FAST ’18), 2018, pp. 265–278.

[11] A. Håklí, D. Taibi, and K. Systa, “Towards cloud native continuous delivery: An industrial experience report,” in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 314–320.

[12] Z. Yan, J. Lu, N. Chainani, and C. Lin, “Workload-aware performance tuning for autonomous dbms,” in 2021 IEEE 37th International Conference on Data Engineering (ICDE), 2021, pp. 2365–2368.

[13] A. S. Higginson, M. Dediu, O. Arsene, N. W. Paton, and S. M. Embury, “Database Workload Capacity Planning using Time Series Analysis and Machine Learning,” in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD ’20), 2020, pp. 769–783.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’12), 2012, pp. 53–64.

[15] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, “Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms,” in Proceedings of the 26th Symposium on Operating Systems Principles (SOSP ’17), 2017, pp. 153–167.

[16] M. C. Calzaraossa, L. Massari, and D. Tessaera, “Workload characterization: A survey revisited,” ACM Computing Surveys (CSUR), vol. 48, no. 3, pp. 1–43, 2016.

[17] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, “Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services,” in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, ser. NSDI ’08. USA: USENIX Association, 2008, pp. 337–350.

[18] Q. Wen, K. He, L. Sun, Y. Zhang, M. Ke, and H. Xu, “RobustPeriod: Time-frequency mining for robust multiple periodicity detection,” in Proceedings of the 2021 International Conference on Management of Data (SIGMOD ’21), 2021, pp. 205–215.

[19] Q. Wen, Z. Zhang, Y. Li, and L. Sun, “Fast RobustSTL: Efficient and robust seasonal-trend decomposition for time series with complex patterns,” in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD ’20), 2020, pp. 2203–2213.

[20] K. Kolos, I. Konstantinou, V. Kantere, and N. Koziris, “Adaptive state space partitioning of markov decision processes for elastic resource management,” in 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 191–194.

[21] A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan, “Peregrine: Workload Optimization for Cloud Query Engines,” in Proceedings of the ACM Symposium on Cloud Computing (SoCC ’19), 2019, pp. 416–427.

[22] G. Yu, P. Chen, and Z. Zheng, “Microscalar: Cost-effective scaling for microservice applications in the cloud with an online learning approach,” IEEE Transactions on Cloud Computing, 2020.

[23] S. El-Beltagy, Y. Mei, L. Cheng, V. Talwar, M. Y. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz et al., “Turbine: Facebook’s service management platform for stream processing,” in 2020 IEEE 39th International Conference on Distributed Computing Systems (ICDCS ’20). IEEE, 2020, pp. 1591–1602.

[24] L. Wang, T. Z. Fu, R. T. Ma, M. Winslett, and Z. Zhang, “Elastictip: Rapid elasticity for realtime stateful stream processing,” in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 573–588.

[25] R. P. Singh, B. Kumarasubramanian, P. Maheshwari, and S. Shetty, “Auto-sizing for stream processing applications at linkedin,” in 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’20), 2020.

[26] J. Jiang, J. Lu, G. Zhang, and G. Long, “Optimal cloud resource auto-scaling for web applications,” in 2013 13th IEEE/ACM Symposium on Cluster, Cloud, and Grid Computing. IEEE, 2013, pp. 274–283.

[27] M. S. Aslanpour, M. Ghorbaci-Arani, and A. N. Toosi, “Auto-scaling web applications in clouds: A cost-aware approach,” Journal of Network and Computer Applications, vol. 95, pp. 26–41, 2017.

[28] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” Journal of grid computing, vol. 12, no. 4, pp. 559–592, 2014.

[29] Y. Al-Dhuraibi, F. Paraiko, N. Djarallah, and P. Merle, “Elasticity in cloud computing: state of the art and research challenges,” IEEE Transactions on Services Computing, vol. 11, no. 2, pp. 430–447, 2017.

[30] T. Chen, R. Babuasoo, and X. Yao, “A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems,” ACM Computing Surveys (CSUR), vol. 51, no. 3, pp. 1–40, 2018.

[31] A. Barnawi, S. Sakr, W. Xiao, and A. Al-Barakati, “The views, measurements and challenges of elasticity in the cloud: A review,” Computer Communications, vol. 154, pp. 111–117, 2020.

[32] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhombinov, “Agrid cold starts for scalable servers,” in 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’19), 2019.

[33] P.-M. Lin and A. Glikson, “Mitigating cold starts in serverless platforms: A pool-based approach,” arXiv preprint arXiv:1903.12221, 2019.

[34] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking behind the curtains of serverless platforms,” in 2018 USENIX Annual Technical Conference (ATC ’18), 2018, pp. 133–146.

[35] M. Calzaraossa and G. Serazzi, “A characterization of the variation in time of workload arrival patterns,” IEEE Transactions on Computers, vol. 34, no. 02, pp. 156–162, 1985.
[39] R. Pitchumani, S. Frank, and E. L. Miller, “Realistic request arrival generation in storage benchmarks,” in 2015 31st Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2015, pp. 1–10.

[40] S. Pacheco-Sanchez, G. Casale, B. Scotney, S. McClean, G. Parr, and S. Dawson, “Markovian workload characterization for qos prediction in the cloud,” in 2011 IEEE 4th International Conference on Cloud Computing. IEEE, 2011, pp. 147–154.

[41] D.-C. Juan, L. Li, H.-K. Peng, D. Marculescu, and C. Faloutsos, “Beyond poisson: Modeling inter-arrival time of requests in a datacenter,” in Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer, 2014, pp. 198–209.

[42] N. Roy, A. Dubey, and A. Gokhale, “Efficient autoscaling in the cloud using predictive models for workload forecasting,” in 2011 IEEE 4th International Conference on Cloud Computing. IEEE, 2011, pp. 500–507.

[43] P. Jamshidi, C. Pahl, and N. C. Mendonça, “Managing uncertainty in autonomic cloud elasticity controllers,” IEEE Cloud Computing, vol. 3, no. 3, pp. 50–60, 2016.

[44] A. Ullah, J. Li, Y. Shen, and A. Hussain, “A control theoretical view of cloud elasticity: taxonomy, survey and challenges,” Cluster Computing, vol. 21, no. 4, pp. 1735–1764, 2018.

[45] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning,” in 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). IEEE, 2019, pp. 329–338.

[46] Y. Garì, D. A. Monge, E. Pacini, C. Mateos, and C. G. Garino, “Reinforcement learning-based application autoscaling in the cloud: A survey,” Engineering Applications of Artificial Intelligence, vol. 102, p. 104288, 2021.

[47] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente, “Efficient resource provisioning for elastic cloud services based on machine learning techniques,” Journal of Cloud Computing, vol. 8, no. 1, pp. 1–18, 2019.

[48] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, “Model-driven optimal resource scaling in cloud,” Software & Systems Modeling, vol. 17, no. 2, pp. 509–526, 2018.

[49] S. Taherizadeh and V. Stankovski, “Dynamic multi-level auto-scaling rules for containerized applications,” The Computer Journal, vol. 62, no. 2, pp. 174–197, 2019.

[50] S.-J. Kim, K. Koh, S. Boyd, and D. Gorinevsky, “ℓ1 trend filtering,” SIAM review, vol. 51, no. 2, pp. 339–360, 2009.

[51] R. J. Hodrick and E. C. Prescott, “Postwar us business cycles: an empirical investigation,” Journal of Money, credit, and Banking, pp. 1–16, 1997.

[52] H. Rue and L. Held, Gaussian Markov random fields: theory and applications. Chapman and Hall/CRC, 2005.

[53] B. Burns, J. Beda, and K. Hightower, Kubernetes: up and running: dive into the future of infrastructure. O’Reilly Media, 2019.

[54] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, “Borg: the next generation,” in Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–14.

[55] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, “Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces,” in 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS). IEEE, 2019, pp. 1–10.