AutoConfig: Automatic Configuration Tuning for Distributed Message Systems

Liang Bao
School of Computer Science and Technology,
XiDian University
Xi'an, ShaanXi, China
baoliang@mail.xidian.edu.cn

Xin Liu
Department of Computer Science,
University of California, Davis
Davis, California, USA
xinliu@ucdavis.edu

Ziheng Xu
School of Computer Science and Technology,
XiDian University
Xi'an, ShaanXi, China
13096956618@163.com

Baoyin Fang
School of Computer Science and Technology,
XiDian University
Xi'an, ShaanXi, China
fby_xdu@163.com

ABSTRACT

Distributed message systems (DMSs) serve as the communication backbone for many real-time streaming data processing applications. To support the vast diversity of such applications, DMSs provide a large number of parameters to configure. However, It overwhelms for most users to configure these parameters well for better performance. Although many automatic configuration approaches have been proposed to address this issue, critical challenges still remain: 1) to train a better and robust performance prediction model using a limited number of samples, and 2) to search for a high-dimensional parameter space efficiently within a time constraint. In this paper, we propose AutoConfig – an automatic configuration system that can optimize producer-side throughput on DMSs. AutoConfig constructs a novel comparison-based model (CBM) that is more robust that the prediction-based model (PBM) used by previous learning-based approaches. Furthermore, AutoConfig uses a weighted Latin hypercube sampling (wLHS) approach to select a set of samples that can provide a better coverage over the high-dimensional parameter space. wLHS allows AutoConfig to search for more promising configurations using the trained CBM. We have implemented AutoConfig on the Kafka platform, and evaluated it using eight different testing scenarios deployed on a public cloud. Experimental results show that our CBM can obtain better results than that of PBM under the same random forests based model. Furthermore, AutoConfig outperforms default configurations by 215.40% on average, and five state-of-the-art configuration algorithms by 7.21%-64.56%.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238175

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; *Software maintenance tools*; • **Computing methodologies → Classification and regression trees**;

KEYWORDS

distributed message system, automatic configuration tuning, comparison-based model, weighted Latin hypercube sampling

ACM Reference Format:
Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. 2018. AutoConfig: Automatic Configuration Tuning for Distributed Message Systems. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238175

1 INTRODUCTION

Distributed message systems (DMSs), such as Kafka[33], RabbitMQ [58], ActiveMQ [2], and RocketMQ [61], have been widely adopted as the underlying communication backbone to support many different real-time stream data processing applications. These applications have vastly diverse characteristics. To support such diversities, DMSs provide a large number of parameters for users to configure. For example, both Kafka and RocketMQ have 200+ parameters that an application can configure [33, 61].

The configuration of these parameters significantly affects the application performance on DMSs [42, 84]. For instance, changing the *batch.size* parameter of Kafka for three different test cases (#3, 4, and 7, see Table 2) from 1 to 60 can result in 5-10 times throughput gain given an application workload, as shown in Figure 1. Such variation is even more significant if the workload is recurring on a daily base, which is likely, especially for realtime reactive applications supported by DMSs. On the other hand, when fixing the value of *batch.size*, changing the *num.network.threads* parameter from 1 to 20 only leads to 8%-50% throughput variation, which means that *num.network.threads* has much less influence on throughput than that of *batch.size*.

Pursuing good performance of different applications on DMSs is non-trivial as DMSs are complex systems with a large number of configurable parameters that control nearly all aspects of their

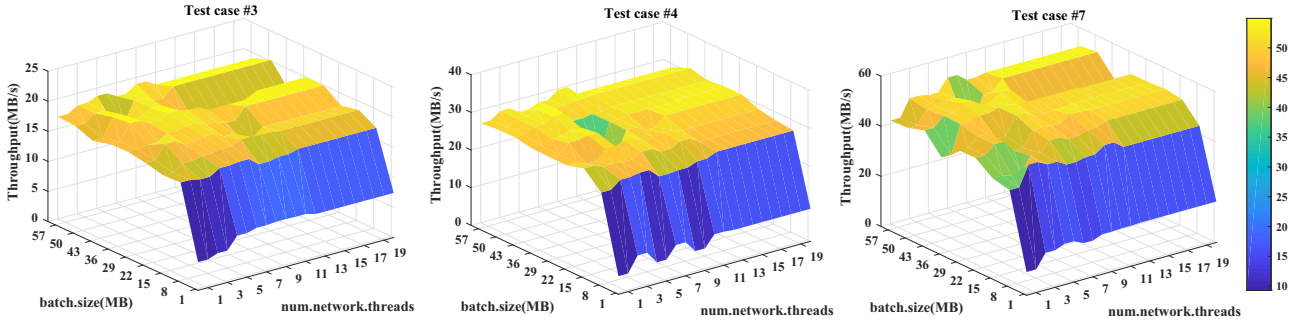

Figure 1: Performance surfaces of three different test cases on Kafka

runtime behaviors. Previous investigations have found that optimizing a DMS to meet the performance requirement of an application has far surpassed the abilities of typical system users [84]. As a result, users often (have to) accept the default settings [42]. Alternatively, many organizations choose to hire expensive experts to configure the DMSs for their applications. Unfortunately, manual configuring is labor-intensive, time-consuming, and often suboptimal. Therefore, there is a dire need for automatical application configurations on DMSs.

Because configuration parameters have high dimensionality, naive exhaustive search is infeasible. Recently, *learning-based* configuration has received much attention. In general, such an approach constructs a performance-prediction model using training samples of different configurations, and then explores better configurations using some searching algorithms. Although previous studies on learning-based configuration have shown promising results, one critical challenge is to construct better and more robust prediction models with a limited number of samples, because generating samples needs to run DMS and is time consuming. Another challenge is to search for more promising configurations in a high-dimensional parameter space.

To overcome these challenges, we propose AutoConfig – an automatic configuration system that aims to configure system parameters for a specific DMS within a given time constraint. AutoConfig consists of three important steps: initial sampling and model training, exploration and exploitation (E&E) process, and best configuration selection. First, we construct a comparison-based model (CBM) from existing samples. The motivation of constructing CMB instead of a conventional prediction-based model (PBM) is that the ultimate goal of configuring a DMS is to find the best configurations rather than to predict the performance of a given system under different configurations. In another word, we care about the relative performance instead of the absolute one. As discussed in Section 4, CBM is more robust than PBM, because CBM uses combinations of the original training samples, which results in a significant larger number of available samples than PBM. Furthermore, AutoConfig searches for better configurations by integrating the CBM and E&E process under the time constraint. The key of the AutoConfig is to train a more robust CBM using combinations of original samples, and to search for promising area of parameter space using skewed random sampling strategy (i.e. wLHS). It also needs to balance the effort on the initial sampling and the E&E process.

In summary, our work makes the following contributions:

- We propose a novel comparison-based performance model (CBM), which is different from the conventional prediction-based model (PBM) used by previous learning-based approaches. CBM allows us to obtain more "combinatorial" training samples from original samples, and thus can produce a more accurate and robust prediction model given a number of original samples.
- We develop the AutoConfig algorithm. It uses weighted Latin hypercube sampling (wLHS) to generate effective samples in the high-dimensional parameter space, and multiple bound-and-search to select promising configurations in the bounded space suggested by the existing best configurations.
- We evaluate the performance of AutoConfig through extensive experiments using eight different testing scenarios in a public cloud. We show that AutoConfig outperforms default configurations by 215.40% on average, and five state-of-the-art tuning algorithms by 7.21%-64.56%.

2 RELATED WORK

DMS configuration has received much attention from both industry and academia. Previous studies on this problem can be classified into four categories: model-based, measurement-based, search-based, and learning-based configuration approaches, as discussed next. The best practices on DMS configuration are also reviewed at the end of this section.

2.1 Model-based Configuration

In model-based configuration, an analytical model is constructed on the early-cycle of the development of a software system [80]. Balsamo et al. [5] reviewed the model-based software performance prediction methods before 2004. Koziolek et al. [38] surveyed the model-based performance evaluation methods for component-based software systems. Commonly used models include queueing network [43, 44, 47], Petri Net [20, 35–37], Palladio Component Model (PCM) [59], and others [23, 27, 48, 56].

Model-based configuration has two main limitations. First, it relies on analytical or design models derived from mathematical theories or software architecture abstraction, which are typically

coarse grained and could be imprecise [13]. Second, model-based configuration cannot adapt as frameworks or hardware evolve. Once the architecture of a specific software system changes, the existing analytical models may not work and need to be redesigned. The same issue arises as clusters evolve with new processors, memory, and storage technologies [77].

2.2 Measurement-based Configuration

Measurement-based configuration aims to evaluate software application focusing on the performance quality features such as response time and throughput. These approaches mainly rely on statistical inferencing techniques to derive performance predictions based on benchmarked measurement data, such as [4, 9, 10, 15, 17, 21, 24, 39, 41, 55, 67, 70, 79, 89]. In this process, many strategies, such as sampling [12, 46, 68], profiling [13, 87], symbolic evaluation [60], and feature interactions [69], are proposed to accelerate performance inference. The prevalent DMS benchmarks are SPECjms2007 [64], jms2009-PS [63], DDSBench [19], and two benchmark suites introduced in [3, 11].

Measurement-based approaches have two main drawbacks. First, they collect program profiles to identify performance bottlenecks, which often fail to capture the overall program performance [13]. Second, most of these approaches lack generality [1], as they are applicable only to specific application scenarios or infrastructures [15, 85].

2.3 Search-based Configuration

Search-based approaches regard configuration problem as a black-box optimization problem and uses search algorithms to solve it, such as in [26, 49, 51, 54, 72, 78]. The objective of the search is to evaluate candidate solutions of different parameters to minimize an objective function. The search strategies include recursive random search (RRS) heuristics [52, 86], evolutionary algorithms [16, 53, 81], hill-climbing algorithms [83], and recursive bound & search [90].

Search-based configuration is simpler and more general compared to other approaches, because it takes the target software system as a black-box function and does not need detailed information about the internals. However, these approaches have to run the application at each iteration of the search, which is time consuming (and impractical) when optimizing large-scale production systems.

2.4 Learning-based Configuration

Most relevant to our work is learning-based configuration. These approaches try to construct performance prediction models first by observing a large collection of running results under different parameter configurations, and then apply some search algorithms to find the optimal configuration based on these models. For example, Sarkar et al. [65] adapted progressive and projective sampling strategies to performance prediction of configurable systems. Guo et al. [22] used CART to predict the performance of configurable systems. Zhang et al. [88] proposed an algorithm based on Fourier learning (FL) for the performance prediction of configurable systems. Sayyad et al. [66] employed a combination of static and evolutionary learning of model structure to find sound and optimum configurations. Soltani et al. [71] employed an artificial intelligence planning technique to automatically select suitable features that satisfy the users' business concerns and resource limitations. Tang [73] provided an approach to find better configurations using random forests model and genetic algorithm based searching strategy. Tantithamthavorn et al. [74] applied Caret method to investigate the performance of defect prediction models. Nair et al. [50] proposed a spectral learning based method to explore the configuration space efficiently to find the measurement that reveal key performance characteristics. Jamshidi et al. [29] proposed a Gaussian processes (GPs)-based method to iteratively capture posterior distributions of the configuration spaces and sequentially drive the experimentation. Bei et al. [6] integrated the random-forest and genetic algorithm to automatically configure the Hadoop configuration parameters for optimized performance for a given application and cluster. Jamshidi et al. [30, 31] conducted an empirical study on four popular software systems to identify the key knowledge pieces that can be exploited for transfer learning. Chen et al. [14] proposed a transfer learning based approach to facilitate the configuration of large-scale computing systems.

Learning-based approaches require considerable numbers of samples to construct a useful performance-prediction model for a software system [51]. This requirement is challenging here because only a limited set of samples can be acquired under a given time constraint, since it is time consuming to sample on a deployed production system. To address this issue, we need to derive a robust prediction model by sampling the high-dimension configuration space wisely, which motivated us to propose a novel comparison-based model and a weighted LHS sampling method.

2.5 Best Practices on DMS Configuration

There have been many best practices devoted to configure DMSs from different perspectives. For example, John et al. [32] focused on two popular DMSs (Kafka and AMQP) and explored the divergence in their features as well as their performance under varied testing workloads. Rubio-Conde et al. [62] compared the performance of two open source middleware, namely ZeroC Ice and AMQP, in the context of medical systems. Dobbelaere et al. [18] conducted a qualitative and quantitative comparison of the common features of the two message systems, namely RabbitMQ and Kafka. Prazeres et al. [57] proposed a message-service oriented middleware named FoT-MSOM for the Fog of Things (FoT) paradigm. Xu et al. [84] provided quantitative evidence for the over-delivered flexibility represented by configuration parameters through the study of the large-scale configuration settings of real users. Le et al. [42] made a thorough evaluation of different configurations and performance metrics of Kafka in order to allow users to avoid bottlenecks, and leverage some good practice for efficient stream processing.

3 PROBLEM STATEMENT

In this paper, we study the configuration problem for distributed message systems (short for *CDMS* problem). As illustrated in Figure 2, a distributed message system (e.g. Kafka, RocketMQ, RabbitMQ, etc.) is often deployed on a cloud environment comprised of a collection of interconnected virtual machines (VMs). It serves as a real-time streaming data pipelines that transfer messages for many systems. Once a DMS has been deployed, the user needs to

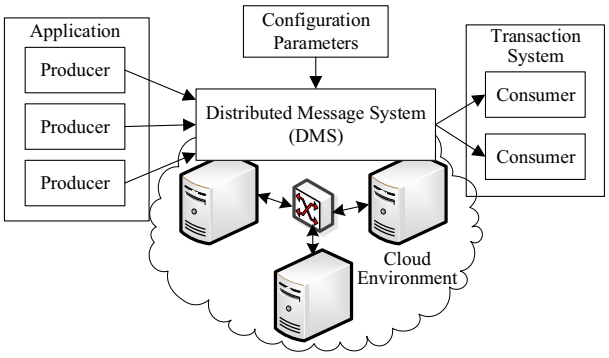

Figure 2: Overview of *CDMS* problem

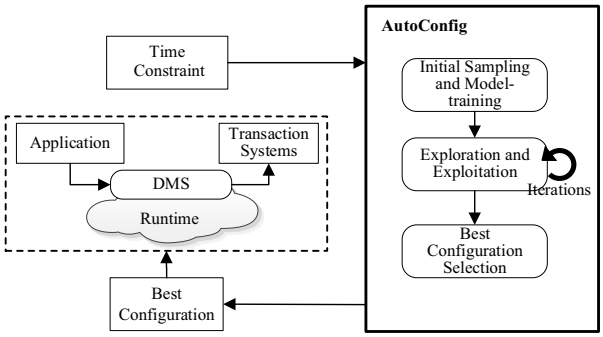

Figure 3: Overview of AutoConfig System

specify its configurations according to the specific application scenario, and such configurations have a significant impact on the performance of the DMS [3, 42, 62, 84].

The goal of *CDMS* problem is to find an optimal configuration that maximizes the (producer-side) throughput of a DMS within a predefined time period, given a specific DMS, an application, and the underlying runtime environment. The reason we choose the *throughput* metric is because it can tell how much data a DMS can store in a unit of time, which characterizes the most important capability of a DMS. Other measurement, such as end-to-end latency (i.e. how long does it take for a message to go though the DMS), can be also introduced as a target metric, because our approach is with generality, and is independent to the metric being optimized. Specially, *CDMS* problem has the following components.

Application: An application represents a real-time streaming data producing system (producers) that wants to transfer messages to transaction systems (consumers). We model it as a 5-tuple A=$\langle l, p, s, r, q\rangle$, where l represents the length of a message; p is the number of producers; s indicates the message sending mode (synchronous or asynchronous); r denotes the message receiving mode (with or without acknowledgement); and q designates the number of available servers (queues).

Runtime Environment: Runtime environment is the execution environment provided to a DMS. We model it as a 5-tuple E=$\langle o, f, m, d, w\rangle$, where o denotes the number of CPU cores; f is the CPU frequency; m represents the physical memory size; d indicates the available disk space; and w reflects the networking setup.

Configuration and Throughput: Let C=$(c_1, c_2, \cdots, c_n)$ be the configuration of a DMS. For example, one can configure as many as 200+ parameters in a Kafka platform, such as the number of network processing threads, the number of I/O processing threads, maximum number of requests in a queue, memory buffer size, etc., as shown in Table 1. Given a configuration C for an application A and its environment E, the throughput is denoted as $TP(A, E, C)$.

Time Constraint: In practice, the time for configuration tuning is often restricted. We define this restricted tuning time as *time constraint*, denoted as TC. Any solution to the *CDMS* problem must terminate when TC is met.

In summary, the *CDMS* problem can be stated as follows:

$$\max_{C \in CB} TP(A, E, C) \tag{1}$$

$$\textit{s.t.}\ \text{tuning time} \leq TC \tag{2}$$

where (1) states that the goal of *CDMS* problem is to find a configuration C that maximizes throughput of a DMS for a given application A and its environment E. In C, the value of each component parameter c_i must be within *CB*, the configuration bound predefined by the DMS. The constraint (2) says that any tuning process of a solution must terminate after a TC amount of time.

4 AUTOCONFIG APPROACH FOR CDMS PROBLEM

The key idea of our approach is to generate a set of samples that is used to train a comparison-based model, and search for more promising configurations using the trained model. Figure 3 illustrates three important steps on our proposed AutoConfig approach.

Initial sampling and model-training. This step generates a set of different configurations using classic Latin hypercube sampling (LHS) method first, given the configuration bound (CB) for each parameter. It then runs the target application on the DMS with these configurations, collect throughput values, and generate the initial training set. Finally, we choose the best of b configurations to generate the initial good configuration set.

Exploration and exploitation. The purpose of this step is to search for better configurations by integrating the comparison-based performance model and the exploration and exploitation (E&E) process under the time constraint. More specifically, we first estimate the weight value for each configuration parameter using Lasso
method and the training set, and initiate our weighted-LHS (wLHS) method. In the exploration phase, we randomly generate a set of configurations using our wLHS within CB to form the exploration configuration set, and train a comparison-based model (CBM) from the training set. In the exploitation phase, we apply a wLHS-based bound and search algorithm (BS) [90] to find potential better configurations near known good configurations using the trained CBM and generate the exploitation configuration set.

Best configuration selection. At each iteration, we update the good configuration set by selecting the best b configurations using

the trained CBM from the current good configuration set, the exploration configuration set, and the exploitation configuration set. If our time constraint (TC) permits more tests, we repeat our exploration and exploitation process. Otherwise, we run the application on the DMS with every configuration in the good configuration set, and return the best one.

4.1 Comparison-based Model

The key step of a learning-based configuration approach is to construct a performance-prediction model for the DMS. Typically, such a model is evaluated based on its accuracy or error. The error percentage can be computed as:

$$MMRE = \frac{|predicted - actual|}{actual} \cdot 100 \quad (3)$$

In this paper, we propose a novel approach that constructs a *comparison-based model* (CBM) which is different from the prediction-based model (PBM) used by most of the previous learning-based approaches as described in Section 2.4. Specially, given the fixed A and E, and two different configurations C_1 and C_2, our model (denoted as $f : C \times C \rightarrow \{0, 1\}$) can compare the throughput values of the DMS with C_1 and C_2:

$$f(C_1, C_2) = \begin{cases} 1, & TP(A, E, C_1) \leq TP(A, E, C_2) \\ 0, & TP(A, E, C_1) > TP(A, E, C_2) \end{cases} \quad (4)$$

There are a number of advantages of using a comparison-based approach:

- *Performance comparison is extremely robust* since it is only mildly affected by outliers [51]. In the context of parameter configuration, a practitioner is often more interested in knowing the rank (by comparison) rather than the predicted performance scores.
- *Performance comparison is the ultimate goal of our CDMS problem.* A user may just want to identify the best configurations rather than to predict the performance of a given system under different configurations. For example, an administrator trying to optimize a Kafka platform is to search for a set of configurations that can obtain maximum throughput, and is not interested in the whole configuration space.
- *Performance comparison increases the number of available training samples significantly.* Suppose we have a set containing n samples by running n different configurations. We can generate $\binom{n}{2} = \frac{n(n-1)}{2}$ samples by simply grouping any two configurations and their throughput values. We will demonstrate in the experiment section that the comparison-based model has better performance than conventional prediction-based approaches, since the number of training samples available to construct a comparison-based approach is increased considerably.

Besides, instead of using residual measures of errors, as described in Equation (3), which depend on residuals ($r = actual - predicte$), we use a *rank-based* measure [50]. Once the comparison-based model is trained, the accuracy of the model is measured by sorting the throughput values of n different configurations, that is:

$$TP(A, E, C_1) \leq TP(A, E, C_2) \leq \cdots \leq TP(A, E, C_n) \quad (5)$$

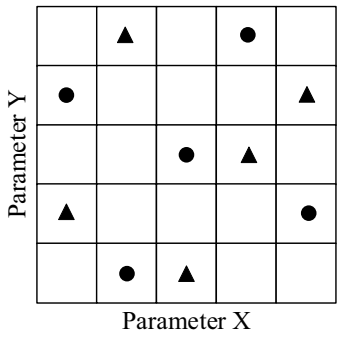

Figure 4: Two sets of LHS samples for a 2D space

Obviously, Equation (5) can be generated by simply using some sort algorithm (e.g. quick sort) and continuously applying our comparison-based model to compare the throughput values between two configurations. After obtaining the predicted rank order, it is compared to the actual rank order. The rank accuracy (RA) is thus calculated using the mean rank difference:

$$RA = 1 - \frac{1}{k} \cdot \sum_{i=1}^{k} \frac{|rank(y_i) - rank(TP(A, E, C_i))|}{n - 1} \quad (6)$$

where n is the number of all configurations, k is number of optimal configurations we want to find from these n candidates. This measure simply counts how many of the pairs in the first k elements of the test data were ordered incorrectly by the prediction model and measures the average of magnitude of the ranking difference.

4.2 Weighted Latin Hypercube Sampling

One important component in our AutoConfig algorithm is the sampling strategy. Because the configuration parameter space is high dimensional, naive sample methods such as random or grid sampling can become very expensive, and is hard to provide a good coverage in it, especially with a small number of samples [86, 90]. To address this issue, we note that the Latin hypercube sampling (LHS) performs better, compared to random or grid sampling, because it allows each of the key parameters to be represented in a fully stratified manner, no matter which parameters are important [45]. Specifically, LHS divides the range of each parameter into k intervals and take only one sample from each interval with equal probabilities [45]. Figure 4 illustrates two sets of LHS samples with five intervals in a 2D dimension, one denoted by dots and the other by triangles.

To make LHS more efficient, we observe that the performance of a DMS is often dominated by a few key configuration parameters (e.g. see Figure 1), and the impact of an influential parameter's values on the performance can be demonstrated through comparisons of performances, regardless other parameters' values [90]. Based on this observation, we proposed weighted LHS (wLHS), which extends the standard LHS algorithm to take into account knowledge about the correlations between configuration parameters and system performance. Such correlations correspond to a linear approximation of the objective function, which can be estimated using the sampled points. The correlation information (i.e. weights) can be

combined with the LHS to generate *skewed random search samples* that are likely to lead to more efficient searches.

Among many models, we adopt linear regression, a simple yet effective statistical method, to measure the strength of the relationship between one or more dependent variables (y) and each of the independent variables (x). These relationships are modeled using a linear predictor function whose weights (i.e., coefficients) are estimated from the data. More specifically, wLHS employs a regularized version of least squares, known as Lasso [75], to estimate the weight for each configuration parameter. Lasso can reduce the effect of irrelevant variables in linear regression models by penalizing models with large weights. The major advantage of Lasso over other methods is that it is interpretable, stable, and computationally efficient [75]. There are also many practical and theoretical studies that have proven its effectiveness as a promising feature selection algorithm [76].

Lasso performs L1 regularization, which adds a penalty equal to the sum of absolute weights times a constant λ to the regression error. This type of regularization can result in sparse models with few nonzero coefficients. Therefore, Lasso regression is able to perform variable selection in the liner model. Variables with nonzero regression coefficients variables are most strongly associated with the response variable. λ is the turning factor that controls the amount of regularization. As the value of λ increases, more coefficients will be set to zero (provided fewer variables are selected) and so among the nonzero coefficients, more shrinkage will be employed. wLHS uses the Lasso path algorithm to determine the order of importance of the configuration parameters of a DMS. The algorithm starts with a high penalty setting where all weights are zero and thus no configurations are selected in the regression model. It then decreases the penalty in small increments, recomputes the regression, and tracks what configurations are added back to the model at each step. wLHS uses the order in which the configurations first appear in the regression to determine how much of an impact they have on the target metric (e.g., the first configuration selected is the most important).

The key ideas in wLHS is that samples should follow the correlation pattern exhibited by the Lasso-based weight calculation. If the past measurements show that smaller values of configuration $c_i \in C$ tend to make the performance better (i.e., a strong positive correlation), then smaller values are more important than larger ones. Hence we should sample more on the smaller values for parameter c_i.

To realize the above weighted sampling idea, we use a truncated exponential density function proposed by Xi [83] for generating the samples. For each component parameter c_i of a configuration ($i = 1, 2, \cdots, n$), we assume a truncated exponential density function of the form:

$$f(c, d, x) = de^{-\omega_i cx} \tag{7}$$

on sampling range $x = [A, B]$, where ω_i is determined by Lasso method and represents the correlation between performance and c_i through all past observations. Parameter c is used to reflect how aggressive the user wants the weighted sampling to be. Parameter d is the normalizing factor so that $f(c, d, x)$ is a density function.

While sampling, we need to divide the interval $[A, B]$ into k intervals with equal probability $1/k$. Let z_j be the j-th dividing point, with $j = 1, 2, \cdots, k$ and $z_0 = A$ and $z_k = B$. Then

$$\frac{j}{k} = \int_{z_0}^{z_j} f(c, d, x)\, dx \tag{8}$$

We now need to draw one point ε_j from given interval $[z_j, z_{j+1}]$ which follows the conditional probability $f(c, d, x) = h$, where h is the scaling constant:

$$1 = \int_{z_j}^{z_{j+1}} \frac{f(c, d, x)}{h}\, dx \tag{9}$$

To draw ε_j, we first draw a random number u, from the uniform distribution in $[0, 1]$ and need to satisfy the following equation:

$$u = \int_{z_j}^{\varepsilon_j} \frac{f(c, d, x)}{h}\, dx \tag{10}$$

Finally, we have:

$$\varepsilon_j = -\frac{log(e^{-\omega_i cz_j} - u(e^{-\omega_i cz_j} - e^{-\omega_i cz_{j+1}}))}{\omega_i c} \tag{11}$$

The solutions for intermediate variables d, z_j, h can be found in [83]. It is worth noting that the weighted LHS takes advantage of knowledge about the correlations between configuration parameters and system performance from past observations. The weight can be based on other metrics (rather than correlation), also one can choose other density functions rather than exponential to realize a similar concept of assigning different weights on different sampling regions [83].

To explain our wLHS method more intuitively, suppose there is a particular parameter c_i with a positive correlation $\omega_i = 0.7$ to the throughput. wLHS that uses the truncated exponential density function (Equation (7)) would divide the sampling space, say [1,100], into 3 equal-probability intervals, as shown in Figure 5. We can observe from Figure 5 that clearly smaller values are stressed more under weighted sampling. The constant $c = 5$ in Equation (7) can be determined through some preliminary studies. Larger c would result in more aggressive weighted sampling [83].

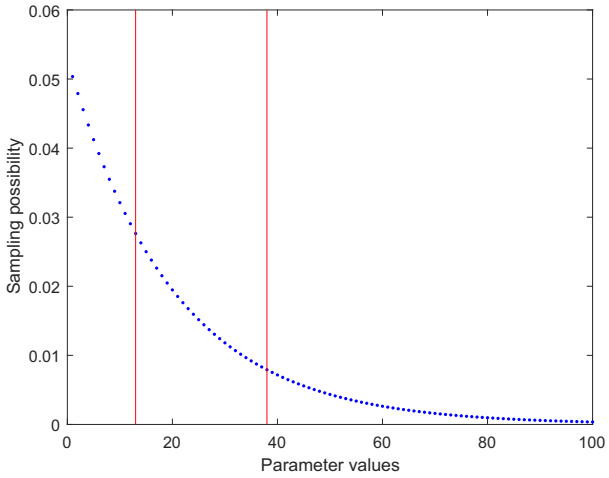

Figure 5: Applying wLHS to a parameter c_i with $\omega_i = 0.7$

4.3 AutoConfig Algorithm

Based on the comparison-based model and the weighted LHS method, we can implement our AutoConfig algorithm. The detailed process is specified in Algorithm 1.

Algorithm 1 ***AutoConfig(DMS,CB,TC)***

Require: *DMS*: the target DMS; *CB*: configuration bounds; *TC*: time constraint.

1: Generate h different configs using LHS within *CB*;
2: Run these h configs on *DMS*, collect throughput results, and generate the initial training set T;
3: $B \leftarrow$ the best b configs in T;
4: **while** *TC* permits more tests **do**
5: Calculate weight set Ω for config parameters using Lasso method and T;
6: Generate h different configs using wLHS and Ω within *CB*, and then choose b out of h configs randomly;
7: Run these b configs on *DMS*, collect throughput results, and generate the exploration set *EP*;
8: $T \leftarrow T \bigcup EP$;
9: Train a comparison-based prediction model M using T;
10: **for** each config $C_i \in B$ **do**
11: Set exploitation set *EI*$\leftarrow \varnothing$;
12: Select h configs using wLHS in the bounded space of C_i;
13: Choose the best config C_i^* from these h configs using M;
14: Run *DMS* with C_i^*, collect the throughput TP_i^*;
15: *EI*=*EI* $\bigcup\{(C_i^*, TP_i^*)\}$;
16: **end for**
17: $B \leftarrow$ the best b configs from $B \bigcup EP \bigcup EI$;
18: $T \leftarrow T \bigcup EI$;
19: **end while**
20: **return** The best config in B;

AutoConfig initially samples h configurations using standard LHS (line 1) and picks b configurations with the best performance (line 2-3). With these h initial samples, we use Lasso method to calculate weight set Ω for configuration parameters (line 5). To explore the configuration space, we apply wLHS with Ω and h intervals, and choose b configurations randomly (line 6). After that, we run these b configurations on the DMS and use the results to generate the exploration set EP (line 7).

Based on already generated samples, we train a comparison-based prediction model based on random forests (line 9), which indicates better performance among many different machine learning algorithms.

To exploit the previously-found best configurations, we apply a bound and search (BS) algorithm [90] to find potential better configurations near already known good configurations (line 10-16). This strategy works well in practice because there is a high possibility that one can find other configurations with similar or better performances around the configuration with the best performance in the sample set [90]. More specifically, for each configuration C_i in B, BS generates another set of samples in the bounded space around C_i: For each parameter c_j in C_i, BS finds the largest value c_j^l (lower bound) that is represented in B and is smaller than that of C_i. It also finds the smallest value c_j^u (upper bound) that is represented in C_i and that is larger than that of C_i. The same bounding mechanism are carried out for every component parameter in C_i. Figure 6 illustrates this bounding mechanism of BS with 2D space (five dots C_1–C_5). After determining the bound for C_i, we use wLHS again to divide each bound into h intervals and generate h samples close to C_i (line 12). Given these h configurations, we use the trained prediction model M to choose the best configuration C_i^* (line 13), and then collect the corresponding throughput TP_i^* by running the DMS with C_i^* (line 14). After that, the sample (C_i^*,TP_i^*) is added to the exploitation set EI (line 15). We repeat these bound and search steps until every configuration in B is tested, and update B with the best b configurations from $B \bigcup EP \bigcup EI$ (line 17). We refine M and Ω by adding new samples from exploration and exploitation phases to the training set T (line 8 and 18).

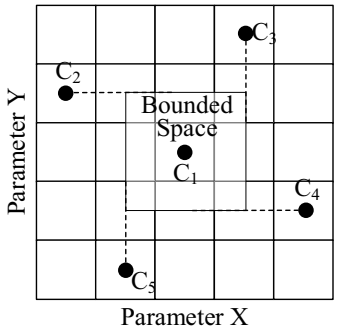

Figure 6: The bound and search (BS) algorithm in a 2D space

Once the time budget on exploration and exploitation is met, we stop searching and return the best configuration in B (line 20).

To satisfy the overall time constraint, we divide the AutoConfig algorithm into two phases, i.e. initial sampling, and exploration and exploitation. Suppose the time constraint is TC, the proportion of time spent in these two phases is denoted as α and β respectively, where $\alpha + \beta = 1$ and $0 \leq \alpha, \beta \leq 1$. Let the average time of running an application with one configuration on the *DMS* equals to t, we have $h \approx \lfloor \frac{\alpha * TC}{t} \rfloor$ and and $b \approx \lfloor \frac{\beta * TC}{2^* iter * t} \rfloor$, where *iter* represent the approximate iterations in the exploration and exploitation phase, and h, b are hyperparameters in AutoConfig algorithm.

5 EXPERIMENTS

We have implemented our approach and conducted extensive experiments under different testing scenarios. The source code can be found in our project website: https://github.com/sselab/autoconfig. In this section, we first describe our experiment setup, and then present the experimental results to prove the efficiency and effectiveness of the proposed approach.

5.1 Experimental Settings

Runtime environment. We conduct our experiments on a public cloud infrastructure named Aliyun[1]. We use 4 Aliyun ECS instances

[1]https://www.aliyun.com

Table 1: Performance-relevant parameters in Kafka

Parameters	Default Value
num.network.threads	3
num.io.threads	8
queued.max.requests	500
num.replica.fetchers	1
socket.receive.buffer.bytes	102400
socket.send.buffer.bytes	102400
socket.request.max.bytes	104857600
buffer.memory	33554432
compression.type	16384
batch.size	0
linger.ms	none

consisting of two types: 1 memory type instance (r5) for producers, and 3 general type instance (g5) for DMS nodes. The producer node is equipped with an 8-core Intel Skylake Xeon Platinum 8163 2.5GHz processor, 16GB RAM, and 50G disk. Each of the DMS nodes is equipped with a 4-core Intel Skylake Xeon Platinum 8163 2.5GHz processor, 8GB RAM, and 50G disk. Both instances have CentOS 6.8 (64bit) installed. All of the VM instances are connected via a high-speed 1.5Gbps LAN.

System and configuration setup. We choose Kafka as our experimental system. Kafka is a distributed streaming platform for publishing and subscribing to streams of records, which is similar to a message queue or enterprise messaging system. We choose Kafka because it is a widely adopted open-source distributed message system.

Based on the Kafka manual [34] and previous studies [18, 42, 82], we identify 11 out of 200+ parameters that are considered critical to the performance of Kakfa platform, as listed in Table 1. Note that even with only 11 parameters, the search space is still enormous, and exhaustive search is infeasible. The reason we choose a small subset of parameters that have great impact on performance instead of all configurable parameters is because reducing the number of tuning parameters can reduce the search space exponentially, and most existing automatic configuration approaches [18, 42, 82] also adopt this feature selection strategy. It's worth noting that because of the linear computing complexity of wLHS on the number of parameters, our approach is general enough and works for high-dimensional parameter spaces.

Benchmark. Based on the previous Kafka testing experiences [25, 40], we design eight testing scenarios with the combinations of different numbers of producers, message sizes, and message acknowledgement modes. Table 2 lists these testing setups.

5.2 Baseline Algorithms

To evaluate the performance of AutoConfig, we compare it with five state-of-the-art algorithms, namely random search [7], BestConfig [90], RFHOC [6], Hyperopt [8], and SMAC [28]. We provide a brief description for each algorithm and report its hyperparameters (if necessary) as follows:

Random search (Random) is a search-based approach that explores each dimension of parameters uniformly at random. It is more efficient than grid search in high-dimensional configuration spaces, and is a high-performance baseline, as suggested in [7].

Table 2: Testing scenarios of our experiment

No.	# of Producers	Message Size	Ack Modes[1]
1	1	0.1KB	-1
2	1	1.0KB	-1
3	1	0.1KB	1
4	1	1.0KB	1
5	3	0.1KB	-1
6	3	1.0KB	-1
7	3	0.1KB	1
8	3	1.0KB	1

[1]In Kafka, Ack=1 means that the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. Ack=-1 means that the leader will wait for the full set of in-sync replicas to acknowledge the record. For more details, see in [34].

BestConfig[2] is a search-based approach that uses divide-and-diverge sampling and recursive bound-and-search algorithm to find a best configuration. We follow the suggestions in [90].

RFHOC is a learning-based approach that constructs a prediction model using random forests, and a genetic algorithm to automatically explore the configuration space. We use the hyperparameters suggested in [6].

Hyperopt[3] is a learning-based approach based on Bayesian optimization. It is widely used for hyperparameter optimization. We use the suggested settings in [8].

SMAC[4] is a learning-based method using random forests and an aggressive racing strategy.

In AutoConfig algorithm, we set the number of iterations to 100 for random forest model, and do not limit the depth of the decision tree and the number of available features for the tree; the proportions of time spent in initial sampling and E&E phases are 0.4 and 0.6 (i.e. α=0.4, β=0.6); the values of other two hyperparameters, i.e. h, and b for each testing scenario are set to 10 and 5, respectively.

For each run in our experiments, every algorithm is executed under the same time constraint and stops once the constraint is met.

5.3 Evaluation Metrics

We consider two performance metrics in our experiments for performance evaluation, namely rank accuracy (RA) and throughput (TP). RA is a metric that evaluates the prediction model, and TP is the ultimate performance metric for AutoConfig.

Rank accuracy (RA). As defined in Section 4, RA is used here to evaluate the quality of a predicted ranking (the prediction) for a set of configurations by comparing it with the corresponding real ranking (the truth). Note again that the ranking-ability of a prediction model is more important than its prediction accuracy in AutoConfig, because we need to use a prediction model to choose the best configurations in each exploitation phase of Algorithm 1 (line 13 in Algorithm 1).

Throughput (TP). TP is the rate of successful message delivery to a distributed message system (i.e. producer-side throughput).

[2]Code is available from: https://github.com/zhuyuqing/bestconf
[3]Code is available from: http://jaberg.github.io/hyperopt/
[4]Code is available from: http://www.cs.ubc.ca/labs/beta/Projects/SMAC

Table 3: The RA values of random forests based CBM and PBM

No.	# of Samples in Training Set	RA (CBM)	RA (PBM)	Best?[1] (CBM)	Best? (PBM)
1	10	0.8592	0.8490	No	No
2	20	0.8469	0.7959	Yes	No
3	30	0.9163	0.7918	Yes	No
4	40	0.9000	0.8653	Yes	Yes
5	50	0.9408	0.9347	Yes	No
6	60	0.9020	0.7020	Yes	Yes
7	70	0.8673	0.8408	Yes	Yes
8	80	0.9224	0.8857	Yes	Yes
9	90	0.9306	0.8327	Yes	Yes
10	100	0.9122	0.8592	Yes	Yes

[1]Whether or not the model can find the best configuration in the test

The TP improvement of an algorithms over the baseline algorithm in comparison is defined as:

$$Imp(baseline) = \frac{TP - TP_{baseline}}{TP_{baseline}} \times 100\%,$$

where $TP_{baseline}$ is the throughput of the baseline, and TP is that of the algorithm being evaluated.

To ensure consistency, we run each application five times and calculate the average of these five runs. The standard deviation of the execution time is 0.02 or smaller, which indicates the stability of the performance.

5.4 Experiment Results

RA. The first experiment is to evaluate the quality of ranking with different prediction models. We first construct a *sample pool* by running Kafka with randomly generated 150 different configurations. After that, 50 samples are selected randomly and then removed from the pool to generate the *testing set*. Last, we randomly choose 10, 20, $\cdots$, 100 samples from the pool to construct 10 different *training sets*. Two random forests based prediction models, one is a comparison-based model (CBM) and another is a prediction-based model (PBM), are trained using these training sets. We then use these trained models to find the best 10 configurations from testing set, respectively. Table 3 lists the RA values on CBM and PBM. We find that all these RA values of CBM outperform those of PBM, and the average improvement is 5.3%. The results also indicate that the CBM can find 9 best configurations out of 10 experiments, where PBM only finds 6 out of 10.

Throughput. Given a fixed time constraint (TC) for each testing scenario, we run six different tuning algorithms plus default configuration independently. Table 4 lists the throughput (MB/s). As expected, the default configuration does not perform well. Our algorithm achieves an average of 215.40% improvement over the default configurations. Furthermore, AutoConfig outperforms all other five algorithms: 11.95%-25.22% improvement over Random, 7.21%-27.45% improvement over BestConfig, 9.52%-38.44% improvement over RFHOC, 10.41%-49.58% improvement over Hyperopt, and 9.66%-64.56% improvement over SMAC.

Finally, we plot the overall throughput improvement percentage of BestConfig, RFHOC, Hyperopt, SMAC and AutoConfig in Figure 7, using the random algorithm as the baseline. In the Figure 7, x-axis lists the eight testing scenarios and y-axis represents the improvement percentage over the random algorithm. We observe that compared with the random algorithm, our approach achieves 11.95%–35.51% improvement among all testing scenarios. AutoConfig achieves an average of 20.78% improvement over Random, 17.71% improvement over BestConfig, 22.03% improvement over RFHOC, 20.52% improvement over Hyperopt, and 23.68% improvement over SMAC. We can conclude from Figure 7 that AutoConfig achieves stable and significant improvements compared with the other five algorithms. Another interesting observation from Figure 7 is that the random search achieves surprisingly good results in our experiments. This is consistent with the findings of Bergstra and Bengio in [7].

5.5 Threats to Validity

Internal validity: To increase internal validity, we performed a controlled benchmark experiment by executing each test case five times and calculate the average of these five runs. Such method can avoid misleading effects of specifically selected test cases and ensures the stability of the throughput result for each scenario. In addition, we use the same sample set and the same random-forest model with identical hyperparameter values to compare the rank accuracy for the CBM and the PBM in each test case. Such results are reliable and can be treated as the ground truth. In our experiments, the value of the time proportion α and β (they control the time ratio between model training and E&E phases, and in turn decide the hyperparameters h and b of AutoConfig) are set to 0.4 and 0.6 respectively, and the value of constant c in Equation (7) is set to 5 (as suggested in [83]). However, the values of these parameters are domain-specific, and can be set by a domain expert. Nevertheless, in cases where the precise values are unknown, using the middle value (for α and β) and the value suggested by previous study [83] (for c) seems to be reasonable choices. In addition, we tried multiple values of these parameters in our experiments and observed that the good values of these parameters that can lead to better throughput results are different from test case to test case.

External validity: We aimed at increasing external validity by choosing eight testing scenarios with various application-level parameters on a mainstream open source software Kafka. Furthermore, we are aware that because the two key components of AutoConfig, i.e. comparison-based model and weighted LHS, are general enough and independent to the system under tune (SUT) [90], the results of our evaluations are transferable to other DMSs.

6 CONCLUSION

In this paper, we propose AutoConfig – an automatic configuration system to optimize throughput for DMSs. AutoConfig constructs a novel comparison-based model (CBM) different from the prediction-based model (PBM) used by previous learning-based approaches. Intensive experiments show that our CBM can obtain better results than that of PBM, under the same random forests based method and a given time constraint. Furthermore, the AutoConfig algorithm uses weighted LHS method to select a set of samples that can provide a better coverage over the high-dimensional parameter space, and searches for more promising configurations

Table 4: Throughput results (MB/s) from different algorithms with fixed time constraints

No.	TC (h)	Default Imp(Default)	Random Imp(Random)	BestConfig Imp(BestConfig)	RFHOC Imp(RFHOC)	Hyperopt Imp(Hyperopt)	SMAC Imp(SMAC)	AutoConfig
1	5	4.28 382.48%	17.28 19.50%	17.46 18.27%	17.57 17.53%	17.25 19.71%	17.41 18.61%	20.65
2	5	7.09 426.80%	30.08 24.17%	29.39 27.08%	28.18 32.54%	24.97 49.58%	28.34 31.79%	37.35
3	5	18.20 47.20%	19.77 35.51%	21.02 27.45%	20.41 31.26%	20.89 28.24%	21.00 27.57%	26.79
4	5	24.85 52.76%	31.21 21.63%	31.53 20.39%	32.80 15.73%	32.65 16.26%	32.56 16.58%	37.96
5	5	6.42 448.60%	31.46 11.95%	31.50 11.81%	25.44 38.44%	31.90 10.41%	30.34 16.08%	35.22
6	5	10.18 293.12%	31.96 25.22%	37.33 7.21%	32.55 22.95%	35.81 11.76%	24.32 64.56%	40.02
7	5	45.5 37.03%	54.87 13.63%	56.27 10.81%	54.31 14.80%	53.96 15.55%	56.86 9.66%	62.35
8	5	68.35 44.59%	86.24 14.60%	85.37 15.77%	90.24 9.52%	83.82 17.91%	82.06 20.44%	98.83

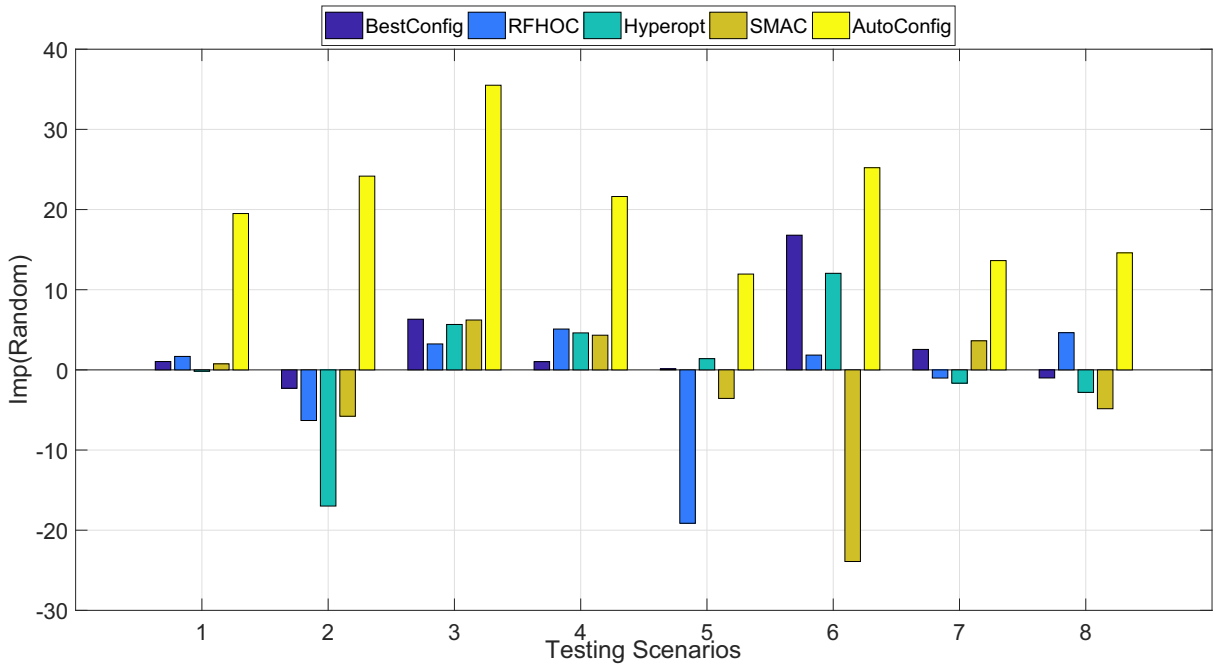

Figure 7: Performance comparison among different algorithms

using the trained CBM. In depth experiments on AutoConfig demonstrate its superior performance to existing five state-of-the-art configuration algorithms on eight different testing scenarios.

Our further work includes refining our AutoConfig approach by supporting the automatic selection of the appropriate hyper-parameters (i.e. α, β, and c) given a specific test case and a time constraint. We will also investigate the possibility of applying our approach to other DMSs such as RabbitMQ, ActiveMQ, and RocketMQ. Last, we hope to abstract the proposed algorithm and release it as an automatic parameter configuring service for other configurable software systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees of ASE'18 for their valuable comments and helpful suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 61202040 with XiDian University and U.S. National Science Foundation under Grant No. CNS-1547461, CNS-1718901, and CCF-1423542 with University of California, Davis. This work is

also supported by the Fundamental Research Funds for the Central Universities (Grant No. JB171005).

REFERENCES

[1] A Abdelaziz, W Kadir, and Addin Osman. 2011. Comparative analysis of software performance prediction approaches in context of component-based system. *International Journal of Computer Applications* 23, 3 (2011), 15–22.

[2] ActiveMQ. 2018. http://activemq.apache.org/.

[3] Sanjay P Ahuja and Naveen Mupparaju. 2014. Performance Evaluation and Comparison of Distributed Messaging Using Message Oriented Middleware. *Computer and information science* 7, 4 (2014), 9.

[4] Stefan Appel, Kai Sachs, and Alejandro Buchmann. 2010. Towards benchmarking of AMQP. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, 99–100.

[5] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. 2004. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering* 30, 5 (2004), 295–310.

[6] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2016. RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1470–1483.

[7] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[8] James Bergstra, Dan Yamins, and David D Cox. 2013. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*. Citeseer, 13–20.

[9] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. 2011. Automated extraction of architecture-level performance models of distributed component-based systems. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 183–192.

[10] Marc Brünink and David S Rosenblum. 2016. Mining performance specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 39–49.

[11] Antonio Carzaniga and Alexander L Wolf. 2002. *A benchmark suite for distributed publish/subscribe systems*. Technical Report. COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE.

[12] Giuliano Casale. 2017. Accelerating performance inference over closed systems by asymptotic methods. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (2017), 8.

[13] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 49–60.

[14] Haifeng Chen, Wenxuan Zhang, and Guofei Jiang. 2011. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Transactions on Knowledge and Data Engineering* 23, 3 (2011), 388–401.

[15] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. 2005. Performance prediction of component-based applications. *Journal of Systems and Software* 74, 1 (2005), 35–43.

[16] Ivan M Delamer, JL Martinez Lastra, and Oscar Perez. 2006. An evolutionary algorithm for optimization of XML publish/subscribe middleware in electronics production. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*. IEEE, 681–688.

[17] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. 2004. Early performance testing of distributed software applications. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 94–103.

[18] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 227–238.

[19] Christian Esposito, Stefano Russo, and Dario Di Crescenzo. 2008. Performance assessment of OMG compliant data distribution middleware. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium On*. IEEE, 1–8.

[20] Stênio FL Fernandes, Wellington João Silva, Mauro JC Silva, Nelson S Rosa, Paulo Romero Martins Maciel, and Djamel Fawzi Hadj Sadok. 2004. On the generalised stochastic petri net modeling of message-oriented middleware systems. In *Performance, Computing, and Communications, 2004 IEEE International Conference on*. IEEE, 783–788.

[21] Krzysztof Grochla, Mateusz Nowak, Piotr Pecka, and Sławomir Nowak. 2017. Influence of Message-Oriented Middleware on Performance of Network Management System: A Modelling Study. In *Multimedia and Network Information Systems*. Springer, 379–393.

[22] Jianmei Guo, Krzysztof Czarnecki, Sven Apely, Norbert Siegmundy, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 301–311.

[23] Jens Happe, Holger Friedrich, Steffen Becker, and Ralf H Reussner. 2008. A pattern-based performance completion for Message-oriented Middleware. In *Proceedings of the 7th international workshop on Software and performance*. ACM, 165–176.

[24] Jens Happe, Dennis Westermann, Kai Sachs, and Lucia Kapová. 2010. Statistical inference of software performance models for parametric performance completions. In *International Conference on the Quality of Software Architectures*. Springer, 20–35.

[25] Gaurav Harsola. 2018. Kafka Benchmarking. https://blog.talentica.com/2016/11/30/kafka-benchmarking/

[26] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining multi-objective search and constraint solving for configuring large software product lines. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 517–528.

[27] Robert Henjes, Michael Menth, and Christian Zepfel. 2006. Throughput performance of java messaging services using websphereMQ. In *Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on*. IEEE, 26–26.

[28] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. *LION* 5 (2011), 507–523.

[29] Pooyan Jamshidi and Giuliano Casale. 2016. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*. IEEE, 39–48.

[30] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 497–508.

[31] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer learning for improving model predictions in highly configurable software. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2017 IEEE/ACM 12th International Symposium on*. IEEE, 31–41.

[32] Vineet John and Xia Liu. 2017. A Survey of Distributed Message Broker Queues. *arXiv preprint arXiv:1704.00411* (2017).

[33] Kafka. 2018. http://kafka.apache.org.

[34] Kafka. 2018. http://kafka.apache.org/documentation/configuration.

[35] Samuel Kounev. 2006. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering* 32, 7 (2006), 486–502.

[36] Samuel Kounev and Christofer Dutz. 2009. QPME: a performance modeling tool based on queueing Petri Nets. *ACM SIGMETRICS Performance Evaluation Review* 36, 4 (2009), 46–51.

[37] Samuel Kounev, Kai Sachs, Jean Bacon, and Alejandro Buchmann. 2008. A methodology for performance modeling of distributed event-based systems. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. IEEE, 13–22.

[38] Heiko Koziolek. 2010. Performance evaluation of component-based software systems: A survey. *Performance Evaluation* 67, 8 (2010), 634–658.

[39] Stephan Kraft, Sergio Pacheco-Sanchez, Giuliano Casale, and Stephen Dawson. 2009. Estimating service resource consumption from response time measurements. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 48.

[40] Jay Kreps. 2018. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). https://events.static.linuxfound.org/sites/events/files/slides/HTKafka2.pdf

[41] Dinesh Kumar, Li Zhang, and Asser Tantawi. 2009. Enhanced inferencing: Estimation of a workload dependent performance model. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 47.

[42] Paul Le Noac'H, Alexandru Costan, and Luc Bougé. 2017. A performance evaluation of Apache Kafka in support of big data streaming applications. In *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 4803–4806.

[43] Yan Liu and Ian Gorton. 2005. Performance prediction of J2EE applications using messaging protocols. In *International Symposium on Component-Based Software Engineering*. Springer, 1–16.

[44] Yan Liu, Ian Gorton, and Alan Fekete. 2005. Design-level performance prediction of component-based applications. *IEEE Transactions on Software Engineering* 31, 11 (2005), 928–941.

[45] Michael D McKay, Richard J Beckman, and William J Conover. 1979. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 21, 2 (1979), 239–245.

[46] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 643–654.

[47] Michael Menth and Robert Henjes. 2006. Analysis of the message waiting time for the fioranoMQ JMS server. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*. IEEE, 1–1.

[48] Gero Mühl, Arnd Schröter, Helge Parzyjegla, Samuel Kounev, and Jan Richling. 2009. Stochastic analysis of hierarchical publish/subscribe systems. In *European Conference on Parallel Processing*. Springer, 97–109.

[49] Alexandr Murashkin, Michał Antkiewicz, Derek Rayside, and Krzysztof Czarnecki. 2013. Visualization and exploration of optimal variants in product line engineering. In *Proceedings of the 17th International Software Product Line Conference*. ACM, 111–115.

[50] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering* (2017), 1–31.

[51] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 257–267.

[52] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 61–71.

[53] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. 2014. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*. ACM, 92–101.

[54] Takayuki Osogami and Sei Kato. 2007. Optimizing system configurations quickly by guessing at the performance. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 35. ACM, 145–156.

[55] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. 2006. Dynamic estimation of cpu demand of web traffic. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. ACM, 26.

[56] Martin Pinzger. 2008. Automated web performance analysis. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 513–516.

[57] Cássio Prazeres, Jurandir Barbosa, Leandro Andrade, and Martin Serrano. 2017. Design and implementation of a message-service oriented middleware for fog of things platforms. In *Proceedings of the Symposium on Applied Computing*. ACM, 1814–1819.

[58] RabbitMQ. 2018. https://www.rabbitmq.com/.

[59] Christoph Rathfelder, Samuel Kounev, and David Evans. 2011. Capacity planning for event-based systems using automated performance predictions. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 352–361.

[60] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. 2010. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 445–454.

[61] RocketMQ. 2018. https://rocketmq.apache.org/.

[62] Paloma Rubio-Conde, Diego Villarán-Molina, and Marisol García-Valls. 2017. Measuring performance of middleware technologies for medical systems: Ice vs AMQP. *ACM SIGBED Review* 14, 2 (2017), 8–14.

[63] Kai Sachs, Samuel Kounev, Stefan Appel, and Alejandro Buchmann. 2009. A Performance Test Harness For Publish/Subscribe Middleware. In *SIGMETRICS/Performance*.

[64] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. 2009. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation* 66, 8 (2009), 410–434.

[65] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-efficient sampling for performance prediction of configurable systems. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 342–352.

[66] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable product line configuration: A straw to break the camel's back. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 465–474.

[67] Abhishek B Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M Voelker. 2008. Automatic request categorization in internet services. *ACM SIGMETRICS Performance Evaluation Review* 36, 2 (2008), 16–25.

[68] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 284–294.

[69] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 167–177.

[70] Julio Sincero, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. 2010. Approaching non-functional properties of software product lines: Learning from products. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 147–155.

[71] Samaneh Soltani, Mohsen Asadi, Marek Hatala, Dragan Gašević, and Ebrahim Bagheri. 2011. Automated planning for feature model configuration based on stakeholders' business concerns. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 536–539.

[72] James Styles, Holger H Hoos, and Martin Müller. 2012. Automatically configuring algorithms for scaling performance. In *Learning and Intelligent Optimization*. Springer, 205–219.

[73] Chong Tang. 2017. System performance optimization via design and configuration space exploration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 1046–1049.

[74] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 321–332.

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

[76] Ryan J Tibshirani, Alessandro Rinaldo, Robert Tibshirani, and Larry Wasserman. 2015. Uniform asymptotic inference and the bootstrap after model selection. *arXiv preprint arXiv:1506.06266* (2015).

[77] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 39–50.

[78] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 455–465.

[79] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. 2012. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 190–199.

[80] Murray Woodside, Greg Franks, and Dorina C Petriu. 2007. The future of software performance engineering. In *2007 Future of Software Engineering*. IEEE Computer Society, 171–187.

[81] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1375–1382.

[82] Jane Wyngaard. 2018. High throughput kafka for science. https://events.static.linuxfound.org/sites/events/files/slides/HTKafka2.pdf

[83] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H Xia, and Li Zhang. 2004. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web*. ACM, 287–296.

[84] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 307–319.

[85] Sherif Yacoub. 2002. Performance analysis of component-based applications. In *International Conference on Software Product Lines*. Springer, 299–315.

[86] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. *ACM SIGMETRICS Performance Evaluation Review* 31, 1 (2003), 196–205.

[87] Sai Zhang and Michael D Ernst. 2014. Which configuration option should i change?. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 152–163.

[88] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance prediction of configurable software systems by fourier learning. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 365–373.

[89] Tao Zheng, C Murray Woodside, and Marin Litoiu. 2008. Performance model estimation and tracking using optimal filters. *IEEE Transactions on software engineering* 34, 3 (2008), 391–406.

[90] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 338–350.