Performance Health Index for Complex Cyber Infrastructures

SANJEEV SONDUR AND KRISHNA KANT*, Temple University, USA

Most IT systems depend on a set of configuration variables (CVs), expressed as a name/value pair that collectively define the resource allocation for the system. While the ill-effects of misconfiguration or improper resource allocation are well-known, there is no effective a priori metrics to quantify the impact of the configuration on the desired system attributes such as performance, availability, etc. In this paper, we propose a *Configuration Health Index (CHI)* framework specifically attuned to the performance attribute to capture the influence of CVs on the performance aspects of the system. We show how *CHI*, which is defined as a configuration scoring system, can take advantage of the domain knowledge and the available (but rather limited) performance data to produce important insights into the configuration settings. We compare the *CHI* with both well-advertised segmented non-linear models and state-of-the-art data-driven models, and show that the *CHI* not only consistently provides better results but also avoids the dangers of a pure data drive approach which may predict incorrect behavior or eliminate some essential configuration variables from consideration.

ACM Reference Format:
Sanjeev Sondur and Krishna Kant. 2021. Performance Health Index for Complex Cyber Infrastructures. 1, 1 (February 2021), 30 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION AND MOTIVATION

1.1 Problem of Configuration Management

As the data centers grow in complexity, sophistication, and size of the infrastructure and services supported, their proper configuration is becoming a huge challenge. Most objects from services down to virtual and physical devices have many configuration parameters (or variables), whose correct setting is crucial for proper functioning and good performance. Many state of art literature [4, 50, 51] highlight that 70%-85% of all users' configuration errors account for the high cost of misconfiguration. Added to this is the poor understanding (and miscommunication) of configuration variables[1] (CVs) on the "outcome" or behavior of the service/system, and hence results in the wrong setting or misconfigured options. Poorly configured systems (or resource allocation) may fail to satisfy the performance, availability, security, and other goals and result in avoidable operational costs and user dissatisfaction. Ill-effects related to system misconfiguration are well documented [40, 51], including their impact on the economy, security incidents [52], service recovery time, loss of confidence, social impact, etc.

In Cloud computing applications, *configuring* the right resources to the Cloud computing objects (i.e. Cloud storage, virtual machines, etc.) is becoming critical, both because of the complexity involved in allocating the right resources and understanding their overall effect on the system (e.g. cost of resource provisioning, user experience, performance,

*Both authors contributed equally to this research.
[1]Most commonly referred to as features [13, 19]

Author's address: Sanjeev Sondur and Krishna Kant, sanjeev.sondur@temple.edu, kkant@temple.edu, Temple University, 1801 N. Broad Street, Philadelphia, PA, USA, 19122.

Manuscript submitted to ACM

energy consumption, etc.) [23, 25, 27]. Further, an application can have complex relations to the resources allocated, the input workload, internal workflows, and the configuration. It is difficult to use straightforward methods to model such relations [2]. *Resource provisioning* for Cloud based applications involves several unique challenges, wherein a Cloud instance is characterized (and priced) based on resource "configuration" (i.e. CPU family/cores, memory, and disk capacity) [25, 46, 47, 53]. Real systems may have 10s to 1000s user-settable configuration variables (or CVs) [21, 49]; in addition, there could be a significant number of hidden or latent manufacturer provided parameters that are not well described. Each of these may take anywhere from two values (for binary variables) to an uncountable number of values, although in practice the feasible values may be limited to either an explicit set of values (e.g., installed memory being one of 32GB, 64GB, 96GB, and 128GB), or approximated by a number of buckets of potentially varying width. Even so, the configuration space (henceforth denoted as Ω) quickly becomes too large to comprehensively characterize it. For example, 10 CVs with 10 values each, amount to 10^{10} or 10B combinations.

1.2 Motivation for Our Work

Thus, we need a more compact way of *understanding the contribution of the individual CVs* on the overall system performance in the context of other settings and to get important insights into the configuration settings. For example, when deciding how much memory to put on a given web server, it is helpful to know roughly at what point the diminishing returns[2] kick in sufficiently strongly to make the additional memory of dubious value. In CherryPick [2], authors noticed that the running time is affected by the amount of resources in the Cloud configuration in a non-linear way. For instance, a regression job on SparkML (with fixed number of CPU cores) sees a diminishing return of running time at 256GB RAM because the job does not benefit from more RAM beyond what it needs. One could ask a similar question regarding the other configuration variables (CVs) such as page size for a database, the local storage allocation for a Cloud Storage Gateway (CSG), CPU usage in a Virtual Machine (VM), etc. Since the main difficulty in evaluating configurations is the interaction among settings of different CVs [21, 44, 46, 49], we need a way of capturing the interactions in a compact manner. Configuration settings representing physical resources (e.g., computing cores, page-size) relate to the cost constraints as well if we open up the possibility of expanding the existing systems; however, we do not address this aspect. In other words, there are predefined ranges for all CVs and any selection must stay within those limits.

It is difficult to get accurate performance data on the Cloud because of various factors [15] like black-box environment, resource contention, performance uncertainty, etc. Further, it is not possible to gather detailed experimental data for all combinations of available configuration parameters. It is for this reason, a **crude** metric such as *Configuration Health Index (CHI)* is valuable wherein, it is important to characterize the effects of a configuration parameter on the observed output based on the limited information available. In theory, we could do an exhaustive set of experiments and perhaps train a neural net for each configuration parameter, however, this is not a feasible methodology due to a large number of configuration parameters involved. Further, we include the results of one such model (section 6) and show such an one-off consideration of the "individual" configuration parameters would loose insights into the important factors affecting the configuration settings.

The work reported here is motivated by our earlier work in *CHeSS* [40] (Configuration Health Scoring System), which defined a *scoring system* to compute the *CHI* for compactly assessing the influence of CVs on various attributes of a system (including performance, availability, security, etc.). We discuss the general *CHI* concept briefly in section 3.1.

[2]The point beyond which, any additional resource allocation is detrimental to performance.

The difference from our earlier work is that, in this paper, we focus entirely on the performance related *CHI* and propose a way of exploiting the limited observational data and the domain expertise to robustly quantify the *CHI* scores. *We validated our approach with relevant data-center configuration experts, who acknowledged that a CHI based approach to provide insights into the configuration settings is a concept way overdue and sorely needed by the industry.*

The involvement of domain knowledge is crucial to make reasonable conclusions from limited data; the technique cannot be purely statistical in this case. While there are risks in using domain knowledge, we mitigate this risk by only expecting the nature of behavior from the domain experts, not its numerical parameters, which are still determined from the limited available data. We show that such an approach is much less risky than a pure data driven approach as the latter could easily lead to misleading conclusions. We are well aware of basic statistical methods of characterizing different factors and their interactions, but generating statistically sound methods for computing confidence intervals or confidence bands with adequate coverage is generally very difficult in such settings [16, 17].

To demonstrate the merits of our approach, we use real world configuration data sets (public domain) from a number of very different systems [28, 36, 37], and our study of the Cloud Storage Gateway [38]. We show that with the limited amount of available data, our method can produce significant insights into the configuration space and produce better results (e.g. health score of CVs, better prediction accuracy, and low variance) since we use data to estimate some key parameters, rather than the actual behavior itself.

The main contributions of this paper are as follows:

- Define an *a priori* mechanism for evaluating the quality of configuration of service in form of a scoring system for its performance.
- Demonstrate how the domain expertise can be exploited to yield more robust score quantification without overburdening the experts.
- Demonstrate that such a scoring system produces important insights into the configuration settings and performs better than the state of the art techniques for a variety of configuration data sets used.

2 CONFIGURATION HEALTH SCORING SYSTEM

We explain the basic concept behind *CHeSS* using Fig 1, where the configuration file is processed by the framework to output a health index of the system. *CHeSS* framework (center box of the same figure) inherently depended on manual input from domain experts to assign the health-scores. The "health" of the system can be characterized along several dimensions (or *attributes*) such as security, availability, manageability, performance, etc., as illustrated on the right side as a 3-D graph in the same figure. For each attribute, we need a measure that is generally considered indicative of that attribute without necessarily having to define a very specific measure, since specificity, while desirable, narrows the applicability of the health measure. The impact of configuration parameters on attributes like availability or security is difficult to access since impact (i.e. behavior) is not readily observable. For example, even if we are allowed to change the length of the security key for testing purposes, it is nearly impossible to determine its security impact experimentally (we can still determine its performance impact).Thus, the motivation for using performance as a key observable metric. In other words, CHI is data enhanced version of CHeSS.

2.1 Quantification of Health Score

The scoring system must provide a compact characterization of the influence of *configuration variables (or CVs)* on various attributes of interest. A preliminary scoring system called *CHeSS* is presented in [40]. A score is a number

between some lower bound & upper bound (e.g. $0 \cdots 2$) where a mid-score (e.g. 1.0) corresponds to a nominal (or "average") configuration, the upper bound corresponds to a highly optimized configuration, and the lower bound corresponds to a very poor (but still operational) configuration. The purpose of a scoring system is to rate configurations in terms of a **normalized measure** of each attribute in a simple way in order to assess the health of the system as a function of the configuration parameters. The health index is not synonymous with very specific or detailed measures that require a detailed quantitative model; instead, it is intended as a measure over the configuration space that provides some indication of how good the configuration is. The distinction is subtle. On one hand, we do want the score to reflect a suitable measure of the attribute (e.g., performance measured in terms of throughput, latency, and other important aspects); on the other, reducing the score to be simply a scaled version of the throughput is of little value, since it too will require detailed modeling. We want to avoid the need for detailed modeling in the context of configuration health because not only it requires a very specific measure, but it also is generally intractable due to a large number of configuration parameters, their interdependencies, and their complex influence on the chosen measure.

Because of how the scoring system is targeted, it necessarily carries some level of non-specificity both in the measures and the values. In particular, with a mid-score (e.g. 1.0) considered as a nominal score, it is the significant deviation from the mid-score that is important, not the precise value. The simplicity in defining and evaluating the score is crucial for scalability in dealing with large configuration spaces. Such non-specificity is inherent to any scoring system, in particular, the well-known Configuration Vulnerability Scoring System (CVSS) [10] that has been used by the security community for quite some time and was the origination motivation for *CHeSS*. Note that CVSS scores are assigned entirely manually based on the "domain knowledge" which consists of both observed and expected impact of a security vulnerability.

The concept in *CHI* should not be regarded as yet another performance prediction model or competitor to existing state of art methodologies as listed in section 7.1 . The main goal of the *CHI* is to generate important insights into relationship between the observed behavior (mostly performance in our paper) and configuration objects and express such a score as a health index. As *CHI* is not a performance prediction model, and in absence of any direct literature, we used the "prediction accuracy" of computing HI (hence, indirect observed behavior O) for an unseen set of configurations. We provide such evaluations comparing *CHI* with other related art in the results section.

2.2 Configuration Specification

Configurations (left side of Fig. 1) are generally specified as name-value pairs defined in configuration files and stored in service specific format (e.g. json, xml, text file or local/remote repository). Service functionality is an abstract term that can take various forms based on users or context, either for processing data, securing services, energy consumed, etc. An a priori scoring of the configuration, as envisioned by *CHI*, will aid the user-community (administrators, designers, developers, end-users, etc.) to gain an insight into the strength or weakness of the configuration beforehand, and hence minimize any costly after-facts.

A configuration file for the service includes a set of configuration "objects" and their settings. Configuration objects are often organized as a hierarchy, with a top-level object representing a feature (QoS, VPN, or VLAN in a router) or component (e.g., namenode & datanode in HDFS), with lower level objects breaking it into finer aspects. For example, a router would have top-level objects for configuration settings of Layer2, Layer3, possibly Layer4, Security, Authentication, etc., each of which has further objects down below. For example, the Layer2 setting includes the spanning-tree protocol setting, with various VLAN settings under that, etc. Configuration settings serve a variety of purposes. Many of them are used for purposes other than the "health"; for example, the EXT4 file system has options for

folding the case in directory searches, enabling extended attributes, support for huge files, etc. Others affect the service health directly in terms of some attributes, e.g., enabling encryption that affects security and performance [37]. However, the health impact of many configuration settings is not obvious and requires varying levels of domain knowledge to assess, e.g., in EXT4, enabling metadata checksum increases resilience, and enabling extent trees results in better performance.

As an example, in *CHeSS* [40], we considered routers in a commercial data center with complex and elaborate working configurations. The largest configuration file here had 22,000 lines and operated on an object hierarchy up to 7 levels deep. Thus the exercise provides a good insight into the usefulness of a scoring system for complex systems where detailed observational data is often spotty or simply unavailable (e.g., the impact of key length on security) and detailed quantitative modeling difficult. Given the intricacies of routing protocols and complex features involving VLANs, authentication, etc., we believe that weights assigned by highly experienced administrators can be regarded as good a depiction of "ground truth" as one might reasonably obtain in such an environment.

A configuration file c of a service contains several CVs henceforth denoted as $P_m, 1 \leq m \leq M$. Each CV is a tuple representing the name and value pair $(P : p)$. Depending on the name/value tuple, the configuration object can contribute to one or more health attributes of the service. For example in Table. 2, a configuration file c_4 contains a configuration object $P_1 : \{mem = 32\}$, which states that 32GB of memory is allocated. This statement can contribute towards the performance attribute by factor p_1 and security attribute by factor s_1. Similarly, another configuration object $P_2 : \{cores = 4\}$ may state CPU resource as 4 cores and contribute towards performance attribute as p_2 and availability attribute as a_2. Thus, each configuration object P influences the service behavior and contributes to one or more attributes (denoted as $\vec{h} = \{p, s, a \cdots\}$). The goal of this research to identify these unknowns (i.e. p_1, p_2, s_1, a_2, etc.) based on the observable behavior of the service with the configuration file (e.g. observed performance in Table. 2, O_1=112Kbps).

2.3 Challenges in Assigning Scores

The key problem in defining *CHI* is two fold: (a) estimation of scores (or *CHI* values) for leaf-level objects in the configuration object hierarchy, and (b) composition of the scores along the hierarchy to determine a score of any arbitrary object. Here (a) can range from a direct assignment of a score by a knowledgeable user/administrator up to an entirely automated estimation. We discuss this aspect in some detail, starting with an entirely manual assignment. We also discuss the composition method used in *CHeSS* and continue to use the same here as well.

2.4 How can *CHI* help? Some preliminary work

In *CHeSS*, we focus on *CHI* in general and evaluated a concrete example relative to three attributes, namely availability (A), security (S), performance (P), for a large commercial routing network. Since the impact of configuration parameters on attributes like availability or security is difficult to determine experimentally, the assignment of scores (or weights) to the leaf-level objects was done by experienced router administrators and then aggregated to estimate the weight or score of an object. This was done recursively from leaves to the root, the end result being the overall score for the router. An example of such an aggregated result is shown as a red asterisk point in Fig. 1, to reflect the overall aggregated score of the device. It is highly desired that such an aggregate score (the asterisk point) lie on the top right corner showing highly desirable configuration, i.e high values of $\vec{H}$. However, it is not just the overall score, but intermediate scores that are also important in assessing the quality of the settings.

It is widely understood that the performance does not increase linearly with the resources thrown at the problem [2] because of the various bottlenecks such as queuing, synchronization, and other delays. For example, if we increase CPU speed, the performance does not increase proportionately due to limitations that may range from CPU microarchitecture (e.g., load/store buffers, bubbles in the pipeline, etc.) to caching to memory bandwidth limitations, to IO bottlenecks, etc. Hence, it is important to model the contribution of CVs w.r.t rate of increase plus a point of diminishing return, thus giving us a concave-convex relationship.

There is a fine distinction between the *CHI* approach and plain curve fitting to the data. Instead of simply observing how the data looks like and fitting the best possible curve to it, we start from the domain knowledge end, which tells us about the general nature of the behavior (e.g., monotonic increase with diminishing returns). Such characteristics are well known to domain experts and can be readily specified by them. Since the domain knowledge cannot tell us what the rate parameters should be in such a behavior, we determine those from the data. The key difference is that we expect this behavior to hold substantially beyond the range for which the data is available. In contrast a purely data-driven approach simply follows the data along with its extrapolation beyond the given data range and increasing lack of confidence (i.e., widening confidence intervals).

The Health Index (HI) metric of the configuration file is expressed as a vector of impacted attributes such as: security (S), availability (A), manageability (M), performance (P), and functionality (F). That is,

$$\vec{HI} = \{S, A, M, P, F, \cdots\} \tag{1}$$

Given the weight, we express the Health Index (H) metric of a configuration as a vector of impacted attributes. As shown in Fig 1, the framework takes the configuration file as input, analyses the configuration statements (CVs, aka configuration objects) for their influence on different attributes, and quantifies the H metric at all levels of the object hierarchy. The right hand side of Fig 1 illustrates a sample result pictorially depicting the health of the configuration. Here the vector H consists of only three attributes P, A, S (performance, availability, security), for ease of displaying in a 3D plot. Each axis shows the upper and lower bounds for the respective attributes P, A, S. Each blue dot refers to the health index h_{mn} of a configuration object CV. A point closer to the upper bounds indicates a higher health index, and hence a good configuration or highly desired configuration. One such example could be a security configuration set to high encryption key length, showing the blue dot with a high health index on the security scale (vertical axis). In Fig. 1, different blue dots correspond to different top level configuration objects. The figure clearly shows that some objects are quite poorly configured, especially with respect to availability and security. The red asterisk mark (*) refers to the aggregated health index HI of the configuration (aggregation as explained in section 2.5). Such a depiction indicates the value of *CHI* concept and allows the administrators to focus on objects whose configuration needs to be improved.

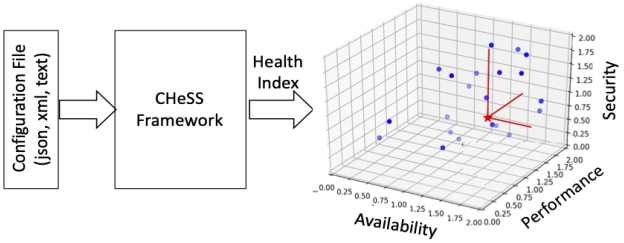


Fig. 1. *CHeSS* Framework [40]

$$\vec{HI} = \{\text{P,A,S}\} = \{1.3,\ 1.2,\ 0.9\}$$

2.5 Aggregation of *CHI* Scores

For the rest of this paper, we will associate $\vec{H}$ with a single attribute representing performance $\mathbb{P}$ [3], and health index ($\vec{h}$) of individual configuration object (P) is marked as a single metric h. The overall metric (or quality of each attribute in H) is represented as the geometric mean of all the contributing attributes (weights) $\vec{h_i}$'s from all the configuration objects P_i. The aggregation was done using simple geometric means since the geometric mean preserves relative scaling and is tolerant of occasional erroneous weight assignment in a large hierarchy. The geometric means provides a measure of the *configuration health index (CHI)* at each level and ultimately for the top-level objects.

The dependencies are considered in the modeling indirectly because *CHI* estimates the HI parameters from the observed behavior. However, the error in the overall estimate (following the geometric mean) is minimized in order to determine the assumed behavioral parameters of all of the CVs (Eq. 2 & 8). This ensures that the dependencies are automatically considered by the *CHI* estimation model. The H_n of the configuration file c_n (and hence the service) is then given as:

$$H_n = \sqrt[M]{\left(\prod_{m=1}^{M}(h_{nm})\right)}$$

or alternatively,

$$log(H_n) = \frac{1}{M}\sum_{m=1}^{M} log(h_{nm}) \tag{2}$$

2.6 Exploiting Domain Knowledge for Performance *CHI*

The key challenge in such an approach *is* to estimate the values of h_i's. With most attributes, including security, availability, manageability, etc., it is generally infeasible to set the CVs to "all" desired values and experimentally determine their impact. Instead, one must estimate the impact via some mathematical model calibrated based on some limited/basic data that might be available. For example, availability (or reliability) modeling generally uses a simple compositional model based on the availability of individual components. Similarly, we may have some quantification of the attack probabilities, which along with suitable attack graph models can give us a quantification of the security of the system. The performance attribute is unique in this respect in that it is possible (at least in theory) to set the CVs to some values and measure the performance (or compute it based on a model calibrated from the observed behavior). This brings in the possibility of at least a partially data-driven determination of the scores, and thereby reduces the amount of manual effort required on part of the domain experts.

However, we cannot immediately swing to the other extreme and claim that there is no need for domain expertise, and everything can be done in a purely data-driven manner. In fact, there are numerous hurdles in making a data-driven approach work and we show in section 5 that it can often lead to misleading results; instead, an approach that judiciously uses expert input can not only improve the quality of the results but also do this with much smaller amounts of data.

2.7 Limitations of Data-driven approach

The key hurdle in a data-driven approach that is often ignored is the difficulty in obtaining adequate quality and quantity of data from a production system. Except in the case of inadvertent mistakes, the configurations that the

[3]Performance is shown as an important attribute by Westermann et al. [48]

administrators are willing to use in a production system are extremely limited – ones that work well. Thus the available data cannot even begin to cover the full range of feasible or even desirable settings. Thus even if we have a huge amount of collected data, its diversity in terms of coverage of the configuration space is extremely limited. Although most production systems do have a small test cluster where any configuration settings are possible, translating either the configuration settings or the results from the test system to the production system (or vice versa) is often either infeasible or involves guesswork (and hence significant errors in the data obtained). Thus the basic requirement of a purely data-driven approach, namely, the ability to generate correct and diverse data covering significant portions of the state space, is usually not met in practice. Unfortunately, the current enthusiasm for applying AI/ML techniques often overshadows these considerations [46].

Even if arbitrary data gathering is permitted (e.g., on a separate cluster), the effort and time required to cover the configuration space make diverse data generation very difficult, as we experienced in our effort to generate CSG data [38]. *This is the main motivation for our performance CHI to be expressed as a scoring system, rather than an exact performance characterization.* It is also the motivation to exploit domain knowledge and use experimental data sparingly rather than following a purely data-driven approach which generally requires extensive amounts of data. Being a coarse-granularity scoring system, the performance *CHI* is concerned with distinguishing, say, a well-performing configuration from a poor one, as opposed to attempting to do a precise estimation of all relevant CVs for near-optimal performance. Nevertheless, for convenience, we view the performance *CHI* as a continuous function of the parameter values and evaluate it using both the available data and the domain knowledge. This allows us to substantially reduce the data requirements and yet obtain much better results than a purely data-driven approach.

The key issue then is how can the domain knowledge be expressed and exploited? It is clear that the input provided by the experts must remain rather small even for large problems. Also, we should not expect experts to provide numbers (e.g., the "weights" as in *CHeSS*) since people tend to make mistakes in providing numbers, and the numbers provided may depend on extraneous factors such as the mood of the person. Instead, we should largely expect experts to provide their insights regarding the system. These insights can often be summarized in the following types of questions:

(1) Based on the knowledge about the system, which CVs are likely to be at least moderately important for deciding the system outcome?
(2) Are certain CVs related by experience based rules of thumb, either precise ones (e.g., each web-server talking to the database needs 10 more DB threads) or fuzzy ones (e.g., each CPU core would add 100-120 MB/s in disk IO requirements)?
(3) Are certain CVs restricted to a certain small set of values (e.g., memory of 32GB, 48GB and 64GB only)?
(4) If the performance generally increases with respect to a CV (e.g., throughput vs. hardware resource amount), is it likely to show a slow decline beyond some point[4] due to increasing overhead or resource contention (we are not asking the expert what that point is)?

The list above is not intended to be comprehensive but will be used in this paper. A similar approach can be used with additional insights.

One concern that always comes up with respect to human involvement is what if the provided insights are incorrect? This can be addressed to some extent by performing sanity checks based on the available data; for example, if we have a decent amount of data, we could do the principal component analysis (PCA) to determine if the importance provided by PCA generally jives with the one provided by the expert. However, we should not lose sight of the fact that a purely

[4]Point of diminishing return

data-driven approach is no panacea, and itself comes with many hazards such as spurious relationships, variations that are opposite to the expected variations, elimination of important variables, overfitting, etc. We demonstrate in this paper that by using the domain knowledge along with the data, we can get more robust results and avoid some of the pitfalls of the pure data-driven approaches.

The key area of our research is to produce important insights of the various CVs (P's) on the observed metric (O's). As CVs can span a wide-dimensional space, a detailed modeling of over 100s of CVs either using a mathematical, simulation, or other technique is laborious (if not impossible) [23, 25, 28, 37, 51]. Further, such one-off models would suffer from robustness and over-fit, i.e. we need to re-do the model for any change in the configuration space.

Authors in Ref [2] support our observation, wherein, the accurate modeling of application performance and building a model that works for a variety of applications and Cloud configurations can be difficult because the knowledge of the internal structure of specific applications is needed to make the model effective. Because of these difficulties, the goal of CHI is to simplify such modeling and discover some important insights into the configuration settings.

2.8 Research Goal

The design goal of *CHI* is to produce a *scoring system*, that can give an insight into the configuration space. That is, *CHI* should: (i) *discover* how a CV influences the behavior (i.e. outcome), (ii) give the rate of increase of such an influence, (iii) show the cut-off point for diminishing returns (if any), and (iv) show the rate of decay beyond the cut-off point. This effect[5] is shown as a convex-concave shaped graph in Fig. 3 below. Instead of building a detailed performance model, the objective is to discover the influence of various CVs on the observable outcome.

With these goals, we formulate the following research questions.

- **(R1)** Discover the influence of the CV P's on the health index h's, including the rate of influence, the point of diminishing return or cut-off point (if any), and the rate of decay (beyond the cut-off point).
- **(R2)** Correlate the H (computed from various h's) of a configuration file to the observed operational metric O (e.g. performance in our case).
- **(R3)** Determine the H^{new} of a new (unseen data) configuration file such that the new H^{new} should reflect the "expected behavior" O^{new} of the new configuration file.

Note that while **(R1)** & **(R2)** are the primary goal of this work, **(R3)** is a natural extension & indirect benefit obtained from discovering the *CHI* metrics. As *CHI* is 'not' a performance prediction model, and in absence of any direct literature, we used the "prediction accuracy" of computing HI in **(R3)** (hence, indirect observed behavior O) for an unseen set of configurations. We compare such metrics with several available literatures *and show that CHI succeeds in characterizing configuration dependence while several of the well advertised methods do not.*

3 SOLUTION DESIGN

The purpose of this research is to get an *apriori* metric to express the health of the configuration file relative to the performance.

3.1 *CHI* Framework

The *CHI* framework to discover the health index (H_n's) of the configuration files (c_n's) is shown in Fig. 2. The configuration objects (P_m's) in the configuration file are first pre-processed and normalized.

[5]The rate of increase plus point of diminishing return

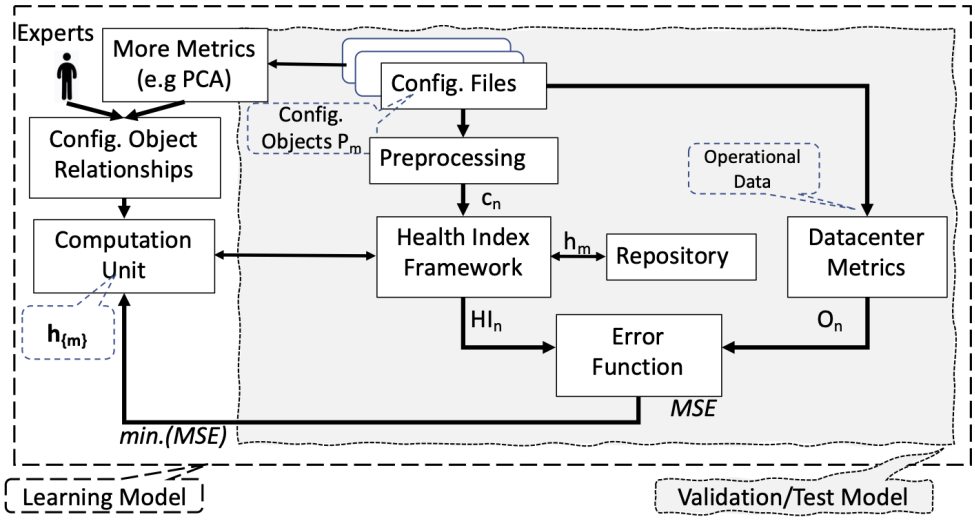


Fig. 2. CHI Framework

Table 1. Abstract Representation of Configuration Files with Geomean (Computed) and Operational Data (Observed)

		Features (Parameter/Value Weights or Config. Objects)									GeoMean	Oper.Data
		P1	P2	P3	P4	P5	P6	P7	P..	P..	HI_n	O_n
Sample Config Files	c_1	p_{mn}	$p_{..}$	$p_{..}$	$p_{..}$	$p_{..}$	p_{mn}	p_{mn}	p_{mn}	p_{mn}	HI_1	O_1
	c_2	$p_{..}$	p_{mn}	p_{mn}	p_{mn}	$p_{..}$	$p_{..}$	$p_{..}$	$p_{..}$	p_{mn}	HI_2	O_2
	c_3	p_{mn}	$p_{..}$	p_{mn}	$p_{..}$	p_{mn}	$p_{..}$	p_{mn}	p_{mn}	$p_{..}$	HI_3	O_3
	c_4	...									HI_4	O_4
	c_5	...									HI_5	O_5

Table. 1 shows an abstract representation of the sample configuration data, with rows illustrating the various configuration files c_n, and columns showing the CVs of the configuration P_m with the respective observed metric O_n. Each cell p_{nm} represent a name/value pair for the configuration files. Table. 2 shows a randomly selected (real-world) example of the CSG configuration file with the associated observed metric (i.e. performance in Bits/sec shown in the last column as O_i). Few attributes (storage IO, metadata size, cache space, metadata space, log space) are enumerated using bucketization shown in Table. 6. The normalized values of the configuration objects and corresponding normalized operational metrics form the basic input to the framework. A sample of a normalized version of the input files is illustrated in Table. 3. Domain experts or service specifications define the boundaries of the configuration object, i.e $P^{(min)}$ & $P^{(max)}$. As part of pre-processing and to keep the format of all input data uniform, it may sometimes be necessary to fill in any undefined/missing values (shown as blank cells in Table. 1). If necessary feature engineering methods have to be incorporated to enrich or supplement an existing feature (i.e configuration object P_m) with a new feature (i.e configuration object P'_m).

3.2 Estimating Health Index From Configuration Data

Our data $\mathcal{D}$ is a set of *distinct* N configuration files, or "rows", say $c_1, .., c_N$ with configuration c_n, $n \in 1..N$ and its corresponding observed output O_n (e.g., performance) (See sample in Table. 1). A configuration is defined by a set of M CVs (or "columns"), denoted $P_1, .., P_M$. That is, each configuration c_n is a vector of M 'values' for CVs $P_1, .., P_M$, henceforth denoted as $p_{n1}, .., p_{nM}$. We postulate the health index H_n for each c_n, which itself is computed as a geometric mean of H of individual CVs (Eq. 2). The H for an individual CV is denoted as h_{nm}, $m = 1..M$, for each CV value p_{nm}.

Table 2. CSG Configuration File from Ref [38] with Operational Data (Observed)

	Features (Parameter/Value Weights or Config Objects)											**Oper.Data**
Config File	Cores	Core Speed	Mem. Size	Mem. BW.	Storage IO	No.of Files	File Size	Metadata Size	Cache Space	MetaData Space	Log Space	Bits/sec.
c_4	4	1.8	32	1.60	3	10000	4KB	2	1	1	2	112166
c_{16}	4	1.8	32	1.60	3	10000	256KB	2	5	3	3	6662268
c_{31}	4	1.8	32	1.60	3	1000	1MB	1	5	3	3	54852372
c_{33}	4	1.8	32	1.60	3	1000	10MB	2	2	2	1	63372836
c_{62}	8	2.1	32	2.10	1	10000	256KB	2	2	1	3	10091093
c_{69}	8	2.1	32	2.10	1	10000	1MB	1	3	2	1	48031772
c_{76}	8	2.1	32	2.10	1	10000	1MB	2	2	1	3	36488192
c_{86}	8	2.1	32	2.10	1	1000	10MB	1	5	3	3	261301724
c_{96}	8	2.1	32	2.10	1	200	1GB	1	3	2	1	304349283

Table 3. CSG Configuration File with Normalized Data

	Features (Parameter/Value Weights or Config Objects) - Normalized											**Oper.Data**
Config File	Cores	Core Speed	Mem. Size	Mem. BW.	Storage IO	No.of Files	File Size	Metadata Size	Cache Space	MetaData Space	Log Space	Bits/sec.
c_4	0.07	0.30	0.33	0.20	0.60	0.80	0.36	0.40	0.00	0.02	0.04	0.50
c_{16}	0.07	0.30	0.33	0.20	0.60	0.80	0.54	0.40	0.13	0.19	0.09	0.68
c_{31}	0.07	0.30	0.33	0.20	0.60	0.60	0.70	0.20	0.13	0.19	0.09	0.77
c_{33}	0.07	0.30	0.33	0.20	0.60	0.60	0.70	0.40	0.01	0.04	0.02	0.78
c_{62}	0.20	0.45	0.33	0.45	0.20	0.80	0.54	0.40	0.01	0.02	0.09	0.70
c_{69}	0.20	0.45	0.33	0.45	0.20	0.80	0.60	0.20	0.02	0.04	0.02	0.77
c_{76}	0.20	0.45	0.33	0.45	0.20	0.80	0.60	0.40	0.01	0.02	0.09	0.76
c_{86}	0.20	0.45	0.33	0.45	0.20	0.60	0.70	0.20	0.12	0.19	0.09	0.84
c_{96}	0.20	0.45	0.33	0.45	0.20	0.46	0.90	0.20	0.02	0.04	0.02	0.85

M	Number of CVs
N	Number of configuration files
P_m	m^{th} CV in a configuration file ($0 \le m \le M$)
c_n	n^{th} configuration file ($0 \le n \le N$)
p_{nm}	name/value pair m of configuration file n
h_{nm}	health index (aka weight) of p_{nm}
H_n	Health Index of n^{th} configuration file
O_n	Observed Metric of n^{th} configuration file
L_{sd}	Strong dependent CVs
L_{wd}	Weakly dependent CVs
L_{un}	Unimportant ones
L	$L = M - L_{sd} - L_{wd} - L_{un}$ dominant CVs
$f_{mk}()$	Relationship function between CVs p_m & p_k
s_{nm}	Normalized value of p_{nm}

Table 4. Nomenclature used in the paper.

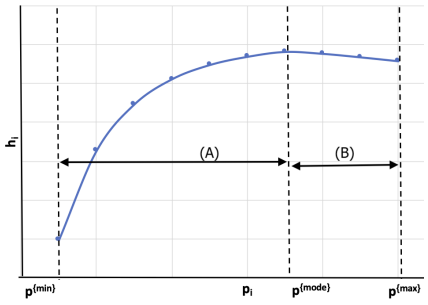


Fig. 3. Sample CV Value vs. *Health Index* relationship

Our goal is to estimate h_{nm}'s, and hence H_ns, compatible with the observed outputs O_n. We pose this as an optimization problem. The assumptions and constraints are as follows.

(I1) Of the M CVs, $L_{sd} < M$ CVs may be *strongly dependent* on others, and we assume that this relationship, denoted as $\models$. Thus, if parm $P_m \models P_k$, then p_{nk} is functionally determined by p_{nm} for all n, i.e., $p_{nk} = f_{mk}(p_{nm})$ where f_{mk} is a known analytic function that transforms column m to column k (independent of the row index n).

(I2) In addition, another $L_{wd} < M$ CVs may be *weakly dependent* on others, and we also assume that this relationship, denoted as $\mapsto$, is given by the experts. Thus, if parm $P_n \mapsto P_k$, then the value p_{nm} restricts the choice of values for p_{nk} to a small range around some value, i.e.,

$$p_{nk} = f_{mk}(p_{nm})(1 + r_k), \qquad r_k \in [-R_k..R_k], k \in \mathcal{L}_{wd} \tag{3}$$

where r_k represents the uncertainty as a fraction. Here $R_k \in 0..1$ is a small (known) fractional number representing the boundaries of the uncertainty. For example, $R_k = 0.1$ means that the value of p_{nk} can vary ±10% around the value determined by the function f_{mk}.

(I3) The relatively important CVs are usually known to the experts from experience or could be obtained using a statistical technique like principle component analysis (PCA). PCA assumes orthogonality and linear combination. *CHI* does not use the direct PCA linear relationship, rather it is used to validate the experts' opinion against PCA results to separate unimportant parameters from important parameters. Others are better eliminated since marginally important CVs only tend to increase the noise in the estimations [39]. We assume that of the $M - L_{sd} - L_{wd}$ primary CVs, L_{un} are unimportant and hence eliminated. Thus, we are left with only $L = M - L_{sd} - L_{wd} - L_{un}$ CVs. The normal $L\ldots$ parameters introduced above represent the sizes of the sets $\mathcal{L}\ldots$ (i.e. $L\ldots = |\mathcal{L}\ldots|$).

(I4) Based on the last few points, we only need to consider L CVs in the formulation. For convenience, we denote the corresponding *set* of variables of different types as $\mathcal{L}_{..}$, i.e., $\mathcal{L}_{sd}$ is set of strongly dependent CVs, $\mathcal{L}_{wd}$ is set of weakly dependent variables, etc.

(I5) We postulate two different forms of functions h_{nm}s that we want to estimate – monotonic and unimodal. Of the L CVs, we assume that L_{mo} is monotonic and L_{um} is unimodal. As before, we represent the corresponding CV sets as $\mathcal{L}_{mo}$ and $\mathcal{L}_{um}$ respectively. This behavior is illustrated as a "convex-concave" graph in Fig. 3, with monotonic behavior depicted in area (A) and unimodal behavior (diminishing returns) beyond point $p^{(mode)}$ in area (B).

As explained earlier, we assume that the minimum and maximum values of the CV, denoted $p_m^{(min)}$ and $p_m^{(max)}$ respectively are defined by experts or available from the experiments. We define $p_m^{(min)}$ as the value for which $h_{nm} = h_{nm}^{(min)}$. Now we have two cases:

Monotonic: h_{nm} increases monotonically with p_{nm} for CV m and when $p_{nm} = p_m^{(max)}$, $h_{nm} = h_{nm}^{(max)}$. We expect the relationship to be concave (i.e., follow the law of diminishing returns). We capture this using the equation:

$$s_{nm} = \frac{p_{nm} - p_m^{(min)}}{p_m^{(max)} - p_m^{(min)}} \tag{4}$$

$$h_{nm} = \frac{1 - e^{-\eta_m s_{nm}}}{1 - e^{-\eta_m}}, \quad m \in \mathcal{L}_{um}, s_{nm}, \quad m \in \mathcal{L}_{mo} \tag{5}$$

where η_m is a predefined positive parameter that controls the growth rate.

Unimodal: Here we assume the same equation as above, except that the maximum happens at the value $p_m^{(mode)} < p_m^{(max)}$. Beyond $p_m^{(mode)}$, we can assume that h_{nm} decreases linearly with maximum fractional degradation of $\gamma_m < 1$. (Generally, $\gamma_m \ll 1$)

$$h_{nm} = \frac{1 - e^{-\eta_m s_{nm}}}{1 - e^{-\eta_m}}, \quad m \in \mathcal{L}_{um}, s_{nm} \leq p_m^{(mode)} \tag{6}$$

$$h_{nm} = 1 - \gamma_m s_{mn}, \quad m \in \mathcal{L}_{um}, s_{nm} > p_m^{(mode)} \tag{7}$$

where

$$s_{nm} = \frac{p_{nm} - p_m^{(mode)}}{p_m^{(max)} - p_m^{(mode)}}$$

Fig. 3 represents the relationship represented in Eq. 5, and Eq. 7 and depicts an example behavior of a CV. h_{nm} for the CV will increase monotonically up to a limit $p^{(mode)}$, and then linearly decreases beyond $p^{(mode)}$. We now have to predict the h_{nm} contribution of the CV p_m given these boundaries. The total number of unknowns is thus $2L + L_{wd} + L_{um}$, and we expect that the number of rows N (i.e., configurations for which output is known) will be significantly larger than the M.

Objective: The objective now is to determine the unknowns introduced above, i.e., $r_k, k \in \mathcal{L}_{wd}$, and $\eta_m, m \in \mathcal{L}_{mo}$ and $\gamma_m, m \in \mathcal{L}_{um}$ to minimize the mean square error (MSE) between the estimated H_n's and observed output O_n's. MSE is given as:

$$MSE = \frac{1}{N} \sum_{n=1}^{N} (H_n - O_n)^2$$

or alternatively,

$$MSE = \frac{1}{N} \sum_{n=1}^{N} (log(H_n) - log(O_n))^2 \tag{8}$$

We discover the individual health index (h_m's) of the configuration objects P_m's to minimize the error between computed H's and observed metric O's (Eq. 8) and thereby determine the η's and γ's as defined in Eq. 5 and Eq. 7. An optimal solution should minimize Mean Square Error (MSE, ideally zero), thereby relating the health index H's as close as possible to the observed metric O's. We denote this estimation error $(H - O)$ as a measure of how far our health index H is from the true *normalized* performance O. In our results, we show the estimation error as the MSE, since it takes into account both the bias and the variance of the estimator [45].

3.3 Computing *CHI*

Fig. 4 and Eqns.(9 – 12) illustrate the concept underlying the computation of *CHI*. With inputs about the CV boundaries ($P^{(min)}$ & $P^{(max)}$) and the pre-processed normalized configuration files, the *CHI* framework in Fig. 2 computes the health index h_{nm} using a non-linear gradient descent regression model to achieve the desired objective (i.e. minimize the MSE). The regression used by *CHI* is intended to estimate the parameters associated with the assumed forms of the functions. MSE is hierarchically dependent on other variables as explained in section 3.2 (item **(I1)** to item **(I5)**). The gradient of MSE (∇MSE) w.r.t individual dependent variable κ_{nm} is represented in Eq. 9, and split into three components: (i) MSE is a function of H (hence, $\partial MSE/\partial H$), (ii) H in turn, depends on individual h_{nm} (hence the second part: $\partial H_n/\partial h_{nm}$), and (iii) individual h_{nm} is a function of either η_m or γ_m (hence the final derivative). To minimize MSE, we employ a gradient descent algorithm, with each iteration calculating a new state κ computed as a function of α & ∇MSE as shown in Eq. 11 (where α represents the learning rate).

$$\nabla MSE = \frac{\partial MSE}{\partial \kappa_{nm}} = \frac{\partial MSE}{\partial H_n} * \frac{\partial H_n}{\partial h_{nm}} * \Psi_{nm} \tag{9}$$

$$\text{where } \Psi_{nm} = \begin{cases} \frac{\partial h_{nm}}{\partial \eta_{nm}}, & \text{if } s_{nm} \leq p_m^{(mode)} \\ \frac{\partial h_{nm}}{\partial \gamma_{nm}} & \text{otherwise} \end{cases} \tag{10}$$

$$\kappa_{nm} \leftarrow \kappa_{nm} - \alpha \nabla MSE \tag{11}$$

$$\text{where } \kappa_{nm} = \begin{cases} \eta_{nm} & \text{if } s_{nm} \leq p_m^{(mode)} \\ \gamma_{nm} & \text{otherwise} \end{cases} \tag{12}$$

The *CHI* computation unit in Fig. 4 represents the calculation of the H (Eq. 2). The unit regresses and computes H, ∇MSE, & κ_{nm} (as given above). An error function computes the difference between "computed" H_n and observed operational metric O_n, and updates the ∇MSE & κ_{nm} for the next states (Eq. 11). The algorithm terminates after it reaches a predefined termination condition (either expressed as the number of iterations or on achieving the desired MSE). At termination, the algorithm persists the 'discovered' relationships η_m's & γ_m's of the configuration object P_m's in the local repository and achieves our research goals (**R1)** & (**R2)**. Next, these relationships can be referred to in the future to compute the H^{new} of a "new/unseen" configuration files, giving us results for the goal (**R3)**. In the results section, we show the effectiveness of the algorithm in discovering the influence of P_m's on O_n's and the *computational accuracy of health indices* (i.e. h_m's). We show that *CHI* (i) can discover the unknown's given in Eq. 12 above to satisfy the objective (minimize MSE) and, (ii) that they relate the contribution of various CVs $P : p$ to the health index h of the configuration object (and indirectly to the O's).

3.4 *CHI* Compute Unit Design

The design in Fig. 4 represents the H computation in Eq. 2, with p's representing the configuration object values (p_{nm}) and the weights (h's) are the contributions of the p_{nm} on the health index h_{nm} as defined by Eq. 5, and Eq. 7. In the traditional ML, a neuron computes an estimated output value $\hat{y}$ equal to the weighed (w) sum of the input features (x), i.e. $\hat{y} = 1/M \sum_{m=1}^{M} (w_m.x_m)$. Following similar concepts, we design the *CHI* computation unit to represent the geometric mean H of the configuration objects in a configuration file, i.e. $\hat{y} = \sqrt[M]{\left(\prod_{m=1}^{M} f(x_m)\right)}$, where $f(x_m)$ represents the health index function of the individual CV (c_m) on the outcome. Thus, h_m=$f(x_m)$ is the unknown and needs to be learned.

CHI compute unit *acts solely* on the weights (h_m=$f(x_m)$) which in turn is a function of η_m's & λ_m's. With a *CHI* design as above, the required solution is the estimation of the parameters η_m & γ_m that contribute to the (weight) health index h's such that it minimizes the mean square error (Eq. 8). The transfer function $g(z)$ represents the non-linearity in the model, represented a: $g(z) = max(\varepsilon, z)$ where ε is a small value (10^{-3}) to ensure a small positive gradient. This transfer function [6] [6] ensures that h_m is always positive and allows the complex relationships in the data to be learned.

[6]Referred in the ML literature as Leaky Rectified Linear Unit (Leaky ReLU)

Our approach for computing Eq. 8 closely resembles Greedy Coordinate Descent (GCD), which usually delivers better function values at each iteration in practice, though Eq. 8 comes at the expense of having to compute the full gradient to select the gradient coordinate with the largest magnitude [20, 22]. Although the non-linear gradient descent regression model can be further improved with robust loss function and optimization techniques, our approach did not venture into this area.

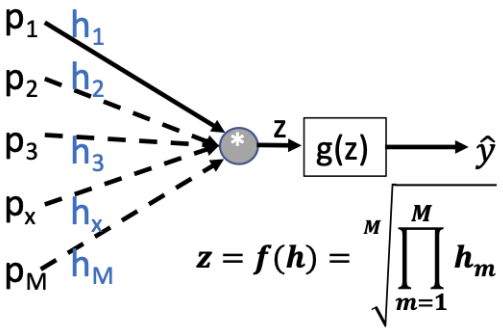


Fig. 4. Compute Unit

3.5 Identifying Unimportant CVs

It is well known that the configuration space is too huge to explore and cover all known combinations of configuration objects. It has been observed that the software performance functions are usually very sparse i.e. only a small number of configurations and their interactions have a significant impact on system performance [14]. Various tools and techniques are being explored to limit such configuration spaces [21, 33]. Most literatures agree that domain expertise is often the best and fastest way to eliminate unwanted features (configuration objects in the problem) [21]. Instead of relying on a pure human approach or trusting a generic algorithm to sort the important and unimportant CVs, our approach eliminates the unimportant CVs (L_{un}) with Principal Component Analysis (PCA) "assisted" domain expertise. PCA is a dimensionality reduction technique that projects the data from its original p-dimensional space to a smaller k-dimensional subspace. By using PCA, domain experts can confirm their belief on which CVs are of importance versus the unimportant CVs (L_{un}). For example, using PCA software-analytics researchers recursively divide data into smaller or as a preprocessor tool to reduce noise in software-related data sets [29, 38, 42].

3.6 *CHI* runtime & Retrain the Model

CHI run time complexity can be expressed as $O(kNM)$ where k is number of iterations, N is number of samples and M is the number of CVs. A coordinate descent method like CHI is of interest due to its simplicity, low cost per iteration, and efficacy [20]. For example, the learning time for CHI ranged between 10 to 25 seconds (Section 4.2.3). As the production system undergoes configuration changes and thereby generates additional data, it is possible to refine the training of the CHI model.

4 EXPERIMENTS AND EMPIRICAL DATA

In this section, we present the data-set used and its characteristics followed by a detailed evaluation of results. All code was developed in Python and all evaluations were run on a MacBook Pro 2.5 GHz x 2 core Intel i7 with 16 GB memory.

4.1 Data Sets and Hyperparameters

A detailed study of system configuration and performance needs a well-defined data-set that captures the resource allocation (i.e. configuration settings) and observed behavior (e.g. performance) under various conditions (e.g. hardware servers, workload, etc.). There are many publicly available data sets as described by Google [35], Alibaba [12], and other Cloud traces [1] capture large time-series data for measures such as CPU utilization, IO rates, network traffic, etc.; unfortunately, they are not useful for us since configuration information is invariably missing. In fact, for some of the data-sets, the configuration continues to change dynamically, but there is no information about it.

We did locate some real world data sets in [28, 36] but the configuration information is not well described, especially for [36]. We also use the CSG dataset that we have created ourselves [38]. Similar to concepts highlighted in CherryPick [2, 24] for Cloud configuration, each of our CSG configurations is represented as the number of CPUs, CPU speed per core, memory, disk speed, and network capacity. CSG data-set has several advantages over other data-sets including (most of all) complete control over and knowledge of the configurations used, data collected, and difficulties encountered. This makes the CSG data much cleaner and usable than others. We obtained the data with some variations in both the hardware setup and the workloads. Note that the hardware setting variations are generally missing from other data sets.

We have no control over the used public data-sets w.r.t data collected, variability in the workload, range of experiments, configuration space explored, etc. and we are limited by whatever data about the HW/SW configuration is included in the dataset. In contrast, our CSG dataset has all of these details. In evaluating *CHI*, we used data-sets that gave us a variety of metrics as observed behavior such as throughput, latency, exec-time, etc. (See Table. 7). The baseline metric used during evaluation is given in the results section.

The usable public data sets that we found for our configuration studies are listed in Table 5. We ran our *CHI* framework on all of these to answer the research questions ((**R1)** to (**R3)**) discussed above. Similar to Ref [2], we use cross-validation, where subsets of the training data can be used to check if the model will generalize well. We followed well established practice similar to an ML approach: the complete data $\mathcal{D}$ (c_n's & O_n's) is first normalized and randomly split into two groups - train ($\mathcal{D}_{train}$) and test ($\mathcal{D}_{test}$). We evaluate using two cases: (i) 50% $\mathcal{D}_{train}$ & 50% $\mathcal{D}_{test}$ and (ii) 80% $\mathcal{D}_{train}$ & 20% $\mathcal{D}_{test}$. The $\mathcal{D}_{train}$ is input to the *CHI* model to compute and discover the unknowns γ_m's, η_m's & h_{nm}'s of various P_m's using the steps explained above. Validating the model against $\mathcal{D}_{test}$ demonstrates the efficacy of the *CHI* and helps to evaluate how the model performs on new/unseen configurations (CVs, hardware, workload, etc.).

Hyper-parameters: To maintain uniformity across all studies and test cases, we maintained the iteration limit (i.e. epochs) to 500 and learning rate α to 0.5 and observed that the *CHI* reaches a satisfactory mean square error (MSE) (i.e min ∇ MSE) during these epochs, and there is no significant improvement afterward. The resulting 'learnt' values of γ's & η's of various P_m's (from the training data $\mathcal{D}_{train}$) is stored in a repository and used to calculate the new health index H_i^{test} of the *unseen test configuration* from $\mathcal{D}_{test}$. We compute the error rate as the difference between computed health index H_i^{test} representing the "expected performance" and observed performance (O's) (as given in Eq. 8). The *error rate* (MSE and variance) of the newly predicted health index (H_i's vs. O_i's) is given in Table 5 for the two split ratios of data-set. Our focus is on understanding the influence of CVs, rather than a performance prediction model, hence we did not venture into detailed ML evaluation metrics such as k-Fold evaluation[7], recall, precision, etc.

Table 5. Data-set used in the paper (and associated *CHI* results).

Code	System [Related Art]	Domain	#Attrs (M)	Samples (N)	A (50/50)		B (80/20)	
					MSE	Variance	MSE	Variance
CSG[a]	Cloud Storage Gateway [38]	Cloud Storage	10	990	0.0121	0.0068	0.0098	0.0058
BB[b]	BitBrains Datacenter [16]	Virtual Machines	7	500	0.0526	0.0256	0.0475	0.0236
SS2[c]	SQL Lite [30]	SQL server	29	2000	0.0620	0.0311	0.0583	0.0308
SS3	Berkeley DB C [29]	Embedded database	18	2000	0.0417	0.0219	0.0332	0.0177
SS8	Apache [37]	Web Server	9	2000	0.0371	0.0212	0.0316	0.0167
SS10[d]	Roll Sort [29]	Sorting Tool	6	3840	0.1887	0.0944	0.1842	0.0932

[a][CSG] https://www.kkant.net/config_traces/CHIproject
[b][BB] http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains (RND500)
[c][SS2,SS3,SS8,SS10] https://github.com/ai-se/ActiveConfig_codebase/tree/master/RawData
[d]~~[SS2,SS3,SS8,SS10]~~ https://goo.gl/689Dve (RawData/PopulationArchives)
[7]Though the above 80/20 test results can represent one of the k-Fold results (for k=5).

Before outlining the evaluation (and for completeness), we briefly describe all the real world data-set used in this paper.

4.2 Data-set Characteristics

4.2.1 Cloud Storage Gateway (CSG) Data-set. CSG is architecturally similar to Edge Computing, IoT Gateways, etc. which are constrained by limited resource capacity and placed between the Edge/IoT/user applications and the Cloud platform. Fig. 5 conceptually shows the CSG operation. A CSG is usually deployed at a branch office or remote location and has access to a rather limited local compute/storage and is connected to a Cloud data center over the Internet. A CSG essentially uses local storage as a cache for the remote Cloud storage to bridge the gap between the demand for low-latency/high-throughput local access and the reality of high-latency connection to the Cloud with unpredictable and usually low throughput. A sample of the design variables in the CSG experiments is shown in Table. 6 and the complete details of experiments, hardware & configuration variation, workloads, and data-sets are given in Ref. [38]. Our study of CSG & the data-set generated is supported by Ref [2], wherein authors state that performance modeling is complex since the running time (performance or throughput) is affected by the amount of resources in the Cloud configuration in a non-linear way and performance under a Cloud configuration is not deterministic.

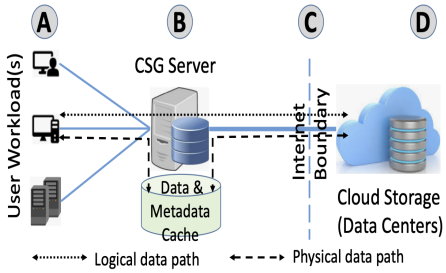


Fig. 5. Edge Computing/ Cloud Storage Gateway

Attribute	No. of Classes	Example of Buckets (Enumeration)
Core Speed (GHz)	5	1.2, 1.8, 2.4 $\cdots$
Memory Capacity (GB)	5	16, 32, 64 $\cdots$
Data cache size (GB)	7	25, 50, 100, 200, 500, 1000, > 1000
Metadata size (GB)	5	25, 50, 100, 200 & 500
Observed Performance	10	Uniform distribution (100Kbps $\cdots$ 350Mbps)

Table 6. Sample of Design Variables for CSG data-set.

The observed performance of CSG denoted as O, is influenced by its configuration variables (CVs), denoted as P_i for ith CV. The CVs include compute resources (cores, cpu-speed, memory capacity, etc.), IO path (memory bandwidth, disk IO bandwidth, etc), buffer space allocation (cache space, meta-data space), etc. We ran about 1000 experiments and collected data on different configurations (denoted $c_n, n = 1, 2, .., N$). Each configuration c_n involves the setting of M different configuration variables (CVs). This data-set was further averaged and smoothed the outliners. Table. 2 illustrates the (randomly selected) configurations c_n's and the corresponding outputs O_n's (known), and the H's (unknown) for such a configuration has to be estimated.

Our CSG configurations include CPU cores, DRAM bandwidth, memory capacity, and storage bandwidth during the execution of workloads (inline with Ref. [23, 24, 45]), although the number of variations that we experimented with had to be limited for practical reasons. Nevertheless, the availability of both hardware and software parameters in our data helps us do a good evaluation and to better explain the results below.

The workload is an important component that defines the behavior of the system and the observable outcome (e.g. performance) [32]. In CSG data-set $\mathcal{D}$, the number of files, file size, and request metadata size refers to the user workload (provided by the vendor). Applying the principles stated in section 3.2, we eliminated the least important CVs (i.e. L_{un} above). For example, using domain knowledge coupled with PCA and reasons explained in the CSG

paper [38], we marked Log Space Resource and Network Bandwidth as unwanted CVs (L_{un}). The normalized data-set of the empirical data is shown in Table. 3. Based on the widespread of a few data-points (e.g. file size, and no. of files), we used Log normalization to re-engineer the configuration object values (p_{nm}) to a new object (p'_{nm}). The full data-set was normalized between a small value ($\varepsilon = 10^{-3}$) and 1.0. After such pre-processing, we use the data-set in the *CHI* to discover individual γ's & η's of various P_m's. With this empirical data in hand, we applied the *CHI* to answer the research questions ((**R1)** to (**R3)**) raised above.

4.2.2 "BitBrains" Data-set. Next, we examine the application of *CHI* to the public domain data from TU Delft BitBrains data-trace [36]. This data-set contains the performance metrics of 1,750 Virtual Machines (VMs) from a distributed data-center from BitBrains, which provides specialized services for managed hosting and business computation for enterprises. This data-set includes some mixture of customer workload from major banks (e.g., ING), credit card operators (e.g., ICS), insurers (e.g., Aegon), etc. During pre-processing, we noticed that the 'fastStorage-1250' data-set contained huge records of zero values (e.g. zero disk IO or network activity) compared to the'Rnd-500' data-set. Therefore, we used the latter, which has 500 VMs that are either connected to the fast SAN (storage area network) systems or to much slower Network Attached Storage (NAS) systems. The data characteristic and usage is described in Table. 7 and the description of the CVs are taken from Ref [36].

4.2.3 "Enterprise" Data-set Characteristics. The last four data sets (SS2, SS3, SS8, SS10) in Table. 5 & 7 have been used in [28, 29, 37] for the performance model, which we compare against our approach. For simplicity, we label these as "Enterprise Data-set". These data sets include traces from a web-server, key-value DBMS, relational DBMS, and a sorting tool. Berkeley DB (C) (marked SS2) is an embedded key-value-based database library that provides scalable high performance database management services to applications. SQLite (SS3) is the most popular lightweight relational database management system used by several browsers and operating systems as an embedded database. Apache HTTP Server (SS8) is a highly popular Web Server. Incidentally, the Apache server has about 550+ [49] CVs[8] but these were cut-down to only nine CVs in [28, 37], but the rationale or the method for doing so is unclear. Roll Sort (SS10) is an environment configuration where "rs" program is run by varying 6 features and the throughput is measured. The characteristics of the Enterprise data-set[9] is given in Table. 7 and the description[10] of the CVs are taken from Ref. [28]. We refer readers to the detailed literature at Ref. [28, 29, 37] for full systems description of the enterprise data sets.

5 DETAILED RESULTS

5.1 Discovering the Influence of configuration objects

Table. 3 shows a small subset of the CSG data-set $\mathcal{D}_{test}$ with all configuration parameters normalized to a range $0 \cdots 1$. Each row represents an input configuration file (c_n's) and the columns correspond to the configuration objects (P_m's). CV names (P's) are given in the header row and the last column refers to the observed output metric (O_n's, in this case, performance expressed as bits/sec).

The results in Table. 8 show the final 'discovered' health index h_{nm}'s in each cell $\{n, m\}$ for various configuration object values p_{nm} (seen in Table. 3) based on the above regression solution. *CHI* computes the γ's & η's for each configuration object P_m's to satisfy the objectives explained earlier and computes the overall health index of the configuration file (H's). The last two columns of Table. 8 show that the computed H's is closely related to the normalized

[8] Apache doc. at: https://httpd.apache.org/docs/2.4/configuring.html & https://httpd.apache.org/docs/2.4/mod/core.html
[9] Data-set at: https://github.com/ai-se/Reimplement/tree/cleaned_version
[10] CV details: http://tiny.cc/3wpwly

Table 7. Characteristics of Data-sets [16, 29, 30, 36, 38]

Code	System	Description of CVs	Observed Behavior
CSG	Cloud Storage Gateway	CSG config.{No.of cores, Core speed, Memory Size, Memory bandwidth, NW bandwidth, Storage IO, Data cache, Meta-data cache, Log space} Workload char. {No.of files, File Size, Meta-data Size}	Completion Time / Performance
BB	TU Delft BitBrains	*VM Cont.Id, Timestamp,* No.of cores, CPU capacity (MHz), *CPU Usage (MHz),* Network Read Bandwidth (KB/s), Network Write Bandwidth (KB/s), Memory Size (MB), *Memory Usage (MB), Memory Usage(%),* Disk Read Throughput (KB/s), Disk Write Throughput (KB/s),	CPU Usage(%)
SS2	SQL Lite server	*OperatingSystemCharacteristics, SQLITESECUREDELETE, ChooseSQLITETEMPSTORE, SQLITETEMPSTOREzero, SQLITETEMPSTOREone, SQLITETEMPSTOREtwo, SQLITETEMPSTOREthree, EnableFeatures, SQLITEENABLEATOMICWRITE, SQLITEENABLESTAT2, DisableFeatures, SQLITEDISABLELFS, SQLITEDISABLEDIRSYNC, OmitFeatures, SQLITEOMITAUTOMATICINDEX, SQLITEOMITBETWEENOPTIMIZATIO0, SQLITEOMITBTREECOUNT, SQLITEOMITLIKEOPTIMIZATIO0, SQLITEOMITLOOKASIDE, SQLITEOMITOROPTIMIZATIO0, SQLITEOMITQUICKBALANCE, SQLITEOMITSHAREDCACHE, SQLITEOMITXFEROPT, Options,* *SetAutoVacuum, AutoVacuumOff, AutoVacuumO0, SetCacheSize, StandardCacheSize, LowerCacheSize, HigherCacheSize, LockingMode, ExclusiveLock, NormalLockingMode, PageSize, StandardPageSize, LowerPageSize, HigherPageSize, HighestPageSize	Performance
SS3	Berkeley DB C	havecrypto, havehash, havereplicatio0, haveverif1, havesequence, havestatistics, diagnostic, pagesize, ps1k, ps4k, ps8k,ps16k, ps32k, cachesize, cs32mb, cs16mb,cs64mb, cs512mb	Performance
SS8	Apache Server	Base, HostnameLookups, KeepAlive,EnableSendfile, FollowSymLinks, AccessLog,ExtendedStatus, InMemor1, Handle	Performance
SS10	Roll Sort	spouts, maxspout, sorters, emitfreq, chunksize, messagesize	Throughput

observed metric (last column O's). During this discovery phase, the minimum MSE achieved was around 0.0128 after 500 iterations.

Table 8. Results: CSG Configuration Files with **computed h_i's &** *HI* metrics

Config File	Cores	Core Speed	Mem. Size	Mem. BW.	Storage IO	No.of Files	File Size	Metadata Size	Cache Space	MetaData Space	HI geoMean	O_i
c_4	0.22	0.71	0.75	0.56	0.91	0.97	0.78	0.94	0.35	0.31	0.58	0.50
c_{16}	0.22	0.71	0.75	0.56	0.91	0.97	0.88	0.94	0.88	0.67	0.70	0.68
c_{31}	0.22	0.71	0.75	0.56	0.91	0.91	0.96	0.96	0.88	0.67	0.70	0.77
c_{33}	0.22	0.71	0.75	0.56	0.91	0.91	0.96	0.94	0.56	0.52	0.65	0.78
c_{36}	0.22	0.71	0.75	0.56	0.91	0.91	0.96	0.94	0.88	0.67	0.70	0.77
c_{62}	0.53	0.85	0.75	0.84	0.53	0.97	0.88	0.94	0.56	0.31	0.68	0.70
c_{66}	0.53	0.85	0.75	0.84	0.53	0.97	0.88	0.94	0.92	0.67	0.77	0.70
c_{76}	0.53	0.85	0.75	0.84	0.53	0.97	0.91	0.94	0.56	0.31	0.68	0.76
c_{86}	0.53	0.85	0.75	0.84	0.53	0.91	0.96	0.96	0.88	0.67	0.77	0.84
c_{96}	0.53	0.85	0.75	0.84	0.53	0.98	0.99	0.96	0.70	0.52	0.74	0.85

After regressing through the data-set to achieve the desired minimum MSE, *CHI* correlates the individual configuration object values p_{nm}'s and their respective h_{nm}'s and determines the "influential behavior" of each of the CVs (P_m's). With the discovered γ's & η's, *CHI* can *build a picture* of how each of these CVs affects the outcome O_n. This relationship is shown in Fig. 6. In this figure, the x-axis shows the normalized values of each CV (shown as the label above sub-graph) and the y-axis is the normalized value of the respective health index (h_i for P_i), and the name of the CV given above the sub-graphs.

These figures demonstrate that *CHI* can discover the behavior with respect to each CV including the strength of the influence, the cut-off point of diminishing return $P_m^{(mode)}$, and rate of decay afterward. The graphical results in Fig. 6 can be visualized by the user to understand how different CVs influence the configuration H (and in turn the service behavior).

We examine these graphical results closely and show that the results are indeed supported by our in-depth study of CSG domain [39]. For example, it is seen that the CSG performance is unimodal with respect to the Cache Space and Meta-Data Space, wherein, there is a threshold beyond which any further increase is detrimental to the system performance. This is supported by our earlier CSG research work [39] wherein we showed that allocating excessive cache space (i.e., blindly throwing resources at the problem) does not help. The CSG needs to perform background tasks such as garbage collection, data eviction to Cloud, data-refresh, etc. Allocating excessive data cache buffer (see sub-graph in Fig. 6) can hurt these background processes, taking additional time to examine the data in the cache and reduce performance. Similar findings on meta-data space configuration is supported by our CSG work in that excessive meta-data space allocation will trigger large metadata operations which in turn takes time, CPU, and memory resources and reduces performance (also observed by Ref. [46]).

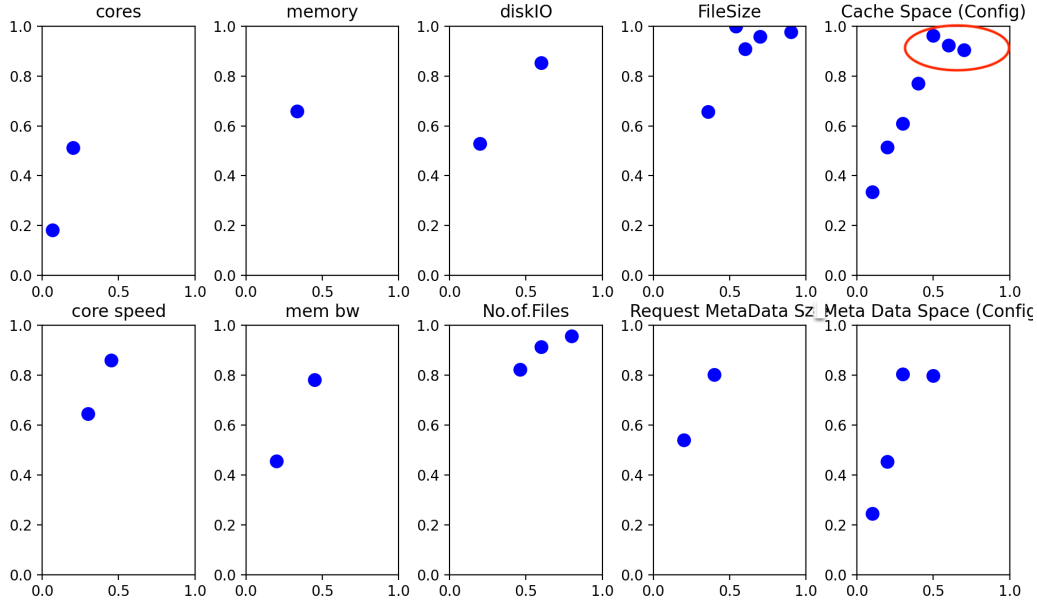


Fig. 6. Results: HI metrics for CSG

5.2 Behavior with "New" Configurations

We use the discovered values, i.e. outcome of the optimization objective (γ_m's & η_m's) to determine the H^{new} of a set of a new (unseen) configuration file. We use the 2^{nd} part of the split empirical data set $\mathcal{D}_{test}$ to validate the *CHI*. The H^{new} is computed using the validation model in the *CHI* framework (marked shaded in Fig. 2). Note that the computation of new H^{new} does not dependent on the compute unit or input from experts or regression logic, because the characteristics of various P_m's is already discovered and stored in the *CHI* repository. Table. 8 shows the computed h_{nm} and H^{new} of the new ($\mathcal{D}_{test}$ configuration files. The last two columns in Table. 8 show that the newly computed H^{new} is closely related to the observed metric O^{new}s (i.e. the true value). This set of results demonstrates that *CHI*

can reasonably determine the probable behavior of the service (i.e observable metric O's) of the new configuration files using the γ's & η's discovered earlier. The MSE and variance (collectively called error rate) for different train/test ratio data-set for various systems is given in Table 5. This also shows that *CHI* can help in evaluating the behavior on a new/unseen configurations (CVs, hardware, workload, etc.).

5.3 *CHI* for "BitBrains" Data

In the absence of an explicit throughput measure, we quantify CPU utilization as an observable metric (O's) and the remaining attributes as CVs (P's). The latter can be changed and allocated differently for the various VMs. In the absence of any further information, we identify each VM as a unique configuration with its associated compute, memory, disk IO, and network resources. We ignore other CVs marked using *italicized font* in row "BB" in Table 7, since these additional CVs represent insignificant aspects of the system. A sample of raw data-set taken directly from Ref. [16] and used in our studies is given in Table. 9. Using this data, we restate the above research question as: *Quantify the influence of various CVs of the VM on the CPU utilization in Bitbrain data-center.*

Since the detailed time-series for each VM setting is not of interest here, we first compute the average value of every parameter for each VM. Given the long length of the trace, the averages should be quite reliable. The results indicate that a few VMs are outliers, with either almost no resource usage in spite of significant resource allocation, or very large resource usage of one type (e.g., VMs that only do very intensive IO). We filtered out all zero value records as this would make the average resource usage so tiny that the entire exercise will be useless. After filtering, we normalized the data-set and used it for input to the *CHI* model. The results are shown in Figs. 7, with each sub-graph showing the influence of a CV on the observable metric. In all the graphs, the x-axis denotes the normalized values of each CV P_i (shown by a label above the graph) and the y-axis is the normalized value of the chosen output metric (O_i), namely the CPU utilization.

Note that in BitBrains data-center architecture, all VMs simply share the available SAN capacity (in terms of disk space and IO throughput), and network capacity. Also, since multiple VMs share the same underlying physical resources, a VM configuration can saturate quickly without yielding additional performance benefits, as the bottleneck can lie elsewhere.

Table 9. Sample FastStorage (RND 500) Configuration File [16]

Identifier	Configurable Variables (CVs)					Neglect		Workload		Observed Metric
Container ID	CPU cores	CPU Capacity [MHZ]	Memory Capacity [KB]	NW Rcvd [KB/s]	NW Trsmt [KB/s]	CPU usage [MHZ]	Memory usage [KB]	Disk read [KB/s]	Disk write [KB/s]	CPU usage [%]
21.csv	8	23408.00	5111808.00	3.16	0.82	392.64	37282.67	93.28	106.89	1.68
108.csv	4	11704.00	16703488.00	175.85	6.97	788.56	494927.40	530.30	1331.83	6.74
136.csv	4	10400.00	1725502.76	29.53	1.68	341.47	1466226.55	220.93	153.90	3.28
392.csv	4	10640.11	16774687.20	243.38	621.50	3507.89	5775129.23	153.33	2914.24	32.97
495.csv	8	20800.00	4173930.42	153.06	68.89	363.93	528867.27	53.81	74.82	1.75

With limited insight into this data-set, we can theorize that performance as a function of the four CVs shown (namely number of CPU cores, memory size, CPU speed, and the disk IO rate) shows a familiar diminishing returns behavior with saturation. This is exactly what we would expect from a basic domain knowledge of computer architecture and IO modeling. For example, the overall CPI (cycles per instruction) for a workload depends on many factors, and thus decreasing only one parameter (e.g., core CPI or access latency) will provide the kind of behavior we see in these graphs.

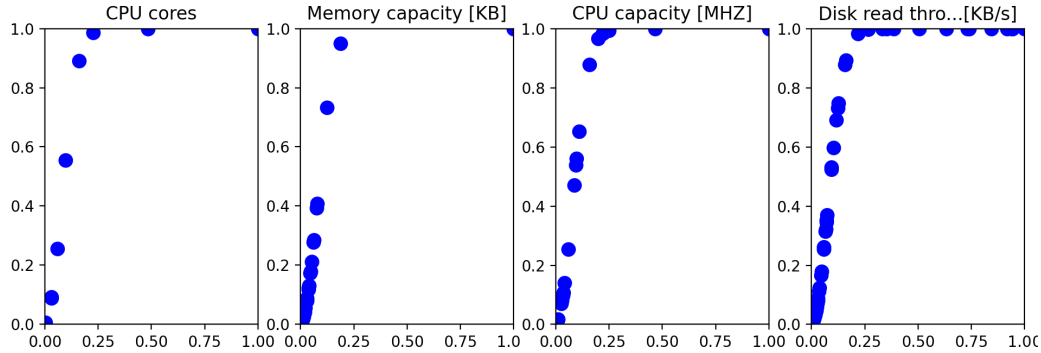


Fig. 7. Results: HI metrics for FastStorage (RND 500)

5.4 *CHI* for Enterprise data sets

The key results from the *CHI* model were summarized earlier in Table 5 (see rows for SS2,SS3,SS8,SS10). With the exception of Roll-Sort, which we discuss shortly, the MSE and its variance are quite low consistently, from about 1.7% to 6.2%. Furthermore, the learning time for *CHI* ranged between 10 to 25 seconds for all these data sets. *These results substantially surpass the prediction results in the literature using these data sets both in terms of accuracy and time.* For example, Ref.[13] uses incremental random samples with steps equal to the number of configuration options (features) of the system. They show rather unstable predictions with a mean prediction error of up to 22%, and a standard deviation of up 46%. Ref. [37] discusses a technique that learn predictors for configurable systems with low mean errors, but the variance in the predictions could be very large; in particular, in half of the results for the Apache Web server predictions, standard deviation was up to 50%. Also, the learning time is reported to be 1-5 hrs depending on the data-set.

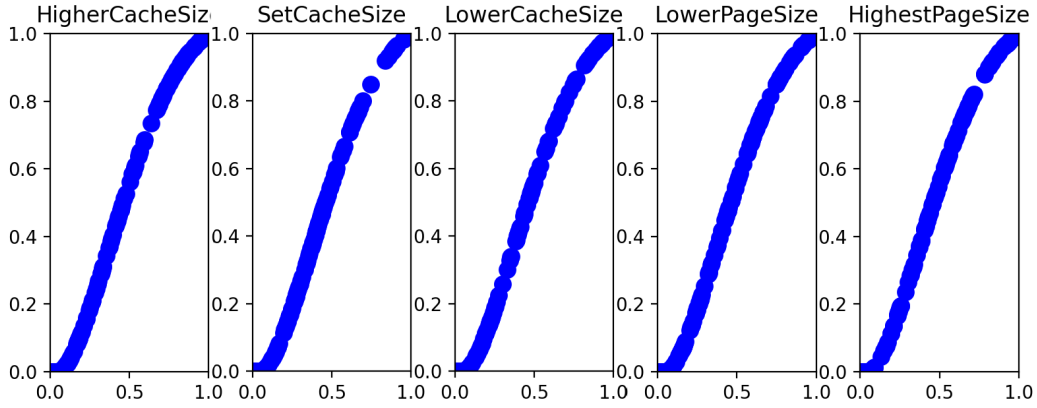


Fig. 8. Results: HI metrics for SQL Lite Configuration

Before discussing the results in Fig. 8 for SQL Lite, we note an important point about its configuration settings. Like most real-world databases, SQL Lite has a large number of configuration parameters, but many of them do not have much influence on the performance. The model used in Ref. [28] had several unexplained options compared to SQL Lite documentation [11]. While we cannot speak directly about this data-set, it appears (based on our deep understanding of how relational databases operate), that these additional parameters (which represent some minor options to be turned

[11] SQL Lite doc. at https://www.sqlite.org/c3ref/c_config_covering_index_scan.html & https://www.sqlite.org/c3ref/c_dbconfig_defensive.html.

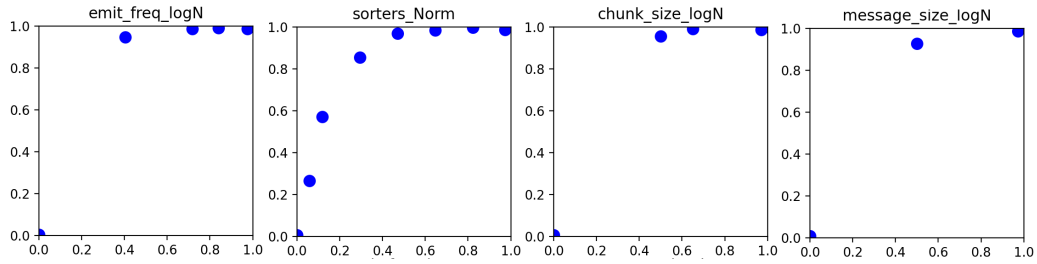


Fig. 9. Results: HI metrics for Roll-Sort Configuration

on/off) should not have a strong influence on SQL-Lite performance. We thus decided to exclude them in our *CHI* modeling is shown in Fig. 8. The excluded CVs are marked as an *italicized* font in Table. 7 and CVs considered by *CHI* is marked as a normal font (after marker *).

Finally, we show the *CHI* models for the Roll-Sort (SS10) workload in Fig. 9. Unlike other workloads, which represent complex applications, Roll-sort is merely a sorting algorithm and has only six CVs, but it is unclear what's special about and whether this is an external sort. The *CHI* model shows that the influence of several CVs saturates at certain values and any further increase in the resource (e.g. No.of sorters, chunk size) does not result in better performance. However, it appears that the data here is very noisy, perhaps influenced by the IO subsystem.

Ref. [28] mentions that for several software systems in their study, the configuration spaces are far more complicated and hard to model. They color code these hard-to-model systems as yellow and red (Fig. 1 in Ref. [28]). Further, they state that applying the state-of-the-art technique by Guo at al. [13] on these software systems showed the error rates of the generated predictor up to 80%. Using the data-set for the same systems used by Ref. [13] (See Table II & III), *CHI* showed a substantial improvement in error rate MSE and variance as shown in Table 10. With 50% $\mathcal{D}_{train}$ & 50% $\mathcal{D}_{test}$ for these data sets, *CHI* achieved an MSE for SQL Lite at 6.2%, for Berkeley DB C: 4.17%, and for Apache Server: 3.71%. *CHI* can *outperform* in most cases since the objective is to discover the influence of individual CVs rather than focus on building a detailed performance model (Table. 10). Additionally, *CHI* does not depend on the sampling techniques which are again data dependent.

6 DISCUSSION: SEGMENTED REGRESSION (MARS & LARS)

The influence of individual CVs on the health index can be complicated [55] and is generally not linear. Yet much of the data-driven behavior, characterization attempts to fit linear or piece-wise linear segments to the observations. In particular, if we have M predictor variables ($X = \{x_i\}$, i $\in 1 \cdots M$) (CVs in our study) and observed output (Y) (performance in our study), a typical assumption is a linear relationship along with a normally distributed error term ε with zero mean and variance σ^2:

$$Y = X\beta + \varepsilon, \text{ where } \varepsilon = \mathcal{N}(0, \sigma^2) \tag{13}$$

Linear Regressions – Ordinary Least Square (OLS), Ridge, Lasso: Such regression algorithms aim to estimate $\hat{\beta}$ (the unknowns) such that some measure of overall error is minimized. For example, OLS regression minimizes the sum of squares of residuals to achieve the unbiased estimate:

$$L_{OLS}(\hat{\beta}) = \sum_{i=1}^{N} (y_i - x_i\beta)^2, \text{ and minimize } \left(\frac{|Y - X\beta|^2}{n} \right) \tag{14}$$

Other algorithms such as Ridge or Lasso regression try to reduce variance at the cost of introducing some bias. For example, Lasso regression adds the constraint $\sum_{j=1}^{M} (|\beta_j| < t)$ where t is a given threshold. Lasso has a parsimony

property [7, 43]: for any given constraint value t , only a subset of the predictor variables (i.e. x_i's) have nonzero values. i.e. many predictor variables *can* have a zero thereby suppressing their contribution to the output Y. That is, in the configuration problem at hand, these algorithms try to "suppress" the contribution of some CVs.

Table 10. Error Rates (MSE and Variance) of Enterprise Data-set

Code	*CHI* Error Rate	Error Rate [13]	Error Rate [37]
[SS2]	6.20% ± 3.11%	7.2% ± 4.2%	N/A
[SS3]	4.17% ± 2.19%	6.4% ± 5.7%	19% ± 1%
[SS8]	3.71% ± 2.1%	9.7% ± 10.8%	27% ± 46%

Further, Ref. [46] argues that machine learning based analytical models, though have been shown to work very well in some specific scenarios, do not consider the domain specific practical factors such as non-linear multi-threading overhead or JVM GC activities, which are very related to soft resource allocation and can significantly degrade server efficiency. Our evaluation supports this statement with empirical results, as given in section 6.1.

Multivariate Adaptive Regression Spline (MARS): MARS [11, 26] is a technique for deriving simple multi-segment models from the data. It can be viewed as an extension of a linear model that automatically models non-linearities and interactions between variables by combining hinge functions of the form $\pm max(0, x - K)$, (where K is a constant). MARS builds a linear model of the form:

$$\hat{y} = \hat{f(x)} = \sum_{i=1}^{k} c_i B_i(x) \tag{15}$$

where the predicted value ($\hat{y}$) is a sum of coefficient (c_i) and basis function ($B_i(x)$). Our investigation revealed that a greedy model like MARS uses brute force to derive the above parts of the model (c_i's & B_i's), and the hinge function cut-off points (K). Though the MARS model can yield good results for predicting new outcomes, an uninformed model like MARS for *CHI* has little regard for the physics of the problem and may behave in unexpected ways such as eliminating certain important CVs or puttin in hinge points (i.e., change in slope) at unexpected places or increase/decrease slope in unexpected ways. For example, instead of showing a steady diminishing-returns property that applies in almost any situation with increasing resources, MARS may as well use a line segment with a larger slope on the higher end!

Least Angle Regression (LARS): LARS [7, 34, 41] produces a full piece-wise linear solution path to a non-linear relationship between predictor variables x_i's and output y. LARS algorithm is similar to forward step-wise regression, but instead of including variables at each step, the estimated parameters are increased in a direction equiangular to each one's correlations with the residual. In section 6.1, we show the limitations of LARS in discovering the influence of the CVs on the performance, wherein the algorithm ignores important CVs though there is a wider variance of such data.

In designing a solution, the "goodness" is often defined in terms of prediction accuracy, but parsimony is another important criterion since simpler models provide better insight into the $X \Rightarrow Y$ relationship [7]. However, we believe that this tradeoff (i.e., more segments implying better accuracy) is introduced somewhat artificially by restricting the model to linear segments which ignore the physics of the problem. Instead, our approach is to find a nonlinear function that shows the desired characteristics (e.g. smoothness, diminishing returns, complexity related loss in performance, etc.) without splitting into more & more segments. We show that such an approach not only correctly captures the expected behavior of the system, it is also less complex.

6.1 Results: Segmented Regression (MARS & LARS)

Although the segmented regression algorithms (MARS and LARS) can do a good job of fitting the data and thereby yield superior prediction accuracy within the range covered by the data, a blind faith in data is particularly troublesome

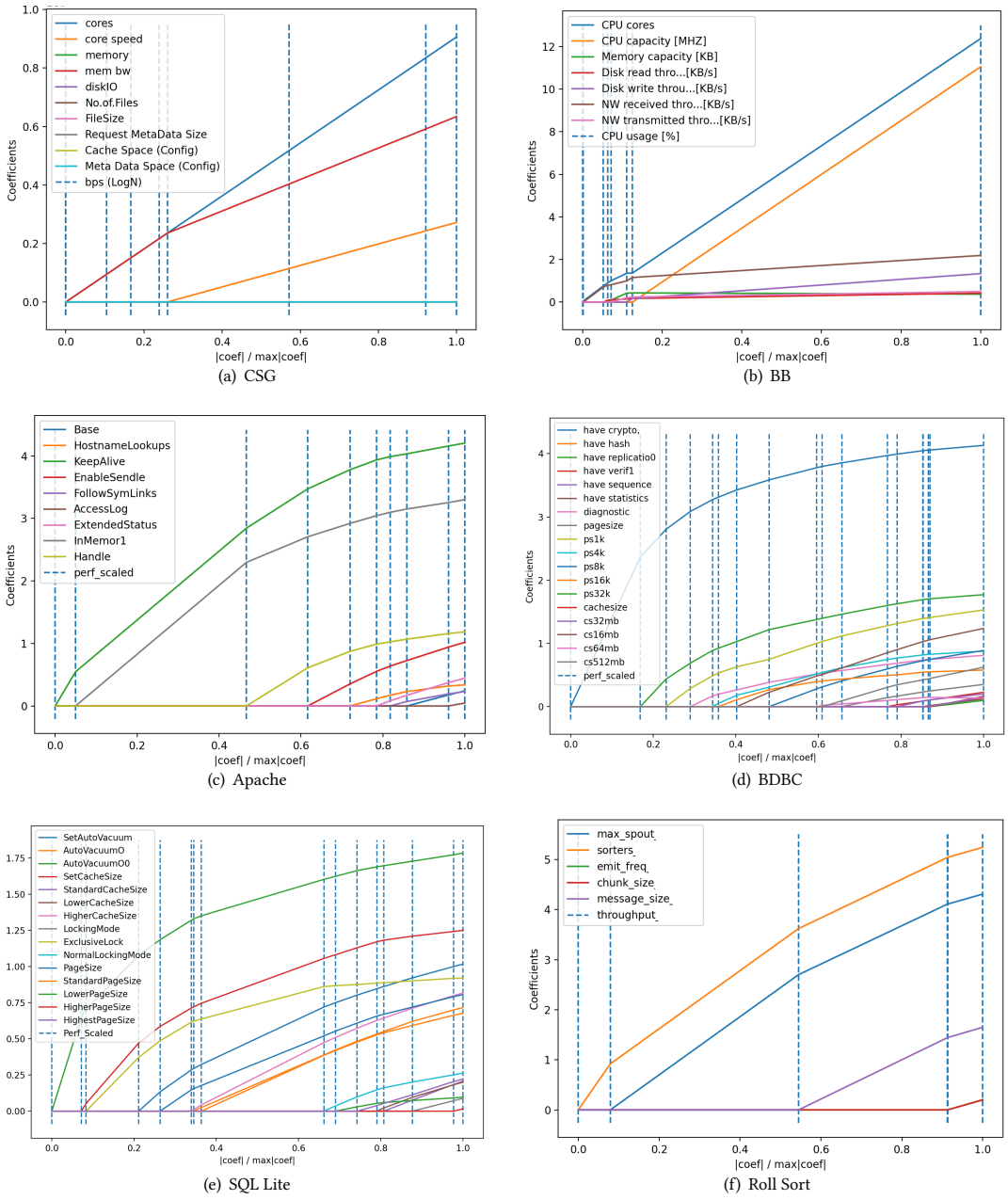
Table 11. Results from Segmented Regression (MARS & LARS) for 80% D_{train} & 20% D_{test}

Code	CVs considered by MARS	CVs ignored by MARS	MARS MSE	LARS MSE
CSG	FileSize, No.of.Files, Cores	Core Speed, Memory, Mem BW, Disk IO, Cache Space, Meta-Data Space, Req.MetaData Size	0.0015	0.0054
BB	NW transmitted, NW received, Memory, Disk Read	CPU cores, CPU capacity, Disk Write,	0.0253	0.0327
SQL Lite	SetAutoVacuum, AutoVacuumO, AutoVacuumO0, SetCacheSize, PageSize, HigherCacheSize, ExclusiveLock, StandardPageSize	StandardCacheSize, LowerCacheSize, LockingMode, NormalLockingMode, LowerPageSize, HigherPageSize, HighestPageSize	0.0262	0.0260
Berkeley DB C	have crypto, diagnostic, ps1k, ps4k, ps8k, ps16k, ps32k, cs16mb, cs512mb	have hash, have replicatio0, have verif1, have sequence, have statistics, pagesize, cachesize, cs64mb	0.0166	0.0170
Apache	EnableSendle, KeepAlive, Handle, InMemor1	Base, HostnameLookups, AccessLog, ExtendedStatus, FollowSymLinks	0.0138	0.0147

for physical systems where we do understand many things about reasonable vs. anomalous behavior. For example, the artificial data fitting by these algorithms often runs counter to sensible behavior, such as showing a higher slope with more resources (i.e., superlinear behavior) where generally one would expect diminishing returns and hence a flattening trend. Even worse, these algorithms may kick out the important CVs and keep the irrelevant ones since they do not have any insight into the nature of individual predictors.

For example, MARS uses a brute force algorithm to regress over a CV P_i to reach the best possible MSE before considering the next CV P_j. This is evident from the results as shown in Table. 11, wherein MARS ignores several CVs for all the domains. In our work involving CSG, we (as experts who have significant experience with it) can confidently say that the ignored CVs (CacheSpace, Meta-DataSpace, Req.MetaData Size, Memory, etc.) have a dominant bearing on the performance of the system. As has been noted in our earlier work [38, 39], although File size and No. of Files are prominent workload characteristics that do have a bearing on the performance, but they are not the primary components that can be isolated from the rest. Similar observations for BitBrains VM components show that MARS ignores most of the CVs and the performance prediction is solely based on two components (Network & Disk). VM performance experts tend to argue that compute capacity (CPU cores, CPU core speed) influences performance heavily.

We show the results from LARS in Fig. 10 for different domains, where the x-axis shows the normalized values of the CV settings (p_i's) and the y-axis shows the normalized values of the performance of the system. Fig. 10(a) is the LARS output for CSG illustrates that performance is heavily dependent on only three components (cores, core speed, memory bandwidth). As systems people, we know that the performance is not dominated by one or two components, but is dependent on a balance between compute, memory, disk IO & workload. Similar results are evident in Fig. 10(b), where the VM performance is linearly dependent on two prime components (CPU cores & CPU capacity[MHz]), largely ignoring the rest of the CVs. This is again in contrast with VM domain knowledge – as basic architecture knowledge would indicate, the compute resource does not have a linear relationship with performance. Instead, the performance depends on the overall CPI (cycles per instruction) which is impacted by cache and memory path latencies. Finally, although MARS & LARS use a similar approach for segmented regression (i,e. converting a non-linear relationship into a series of linear regions), we see that they yield drastically different results, which too is troubling and indicates a dissociation from the physics of the problem.



(a) CSG (b) BB

(c) Apache (d) BDBC

(e) SQL Lite (f) Roll Sort

Fig. 10. Results: LARS Coefficients (weights) for the above data sets

This study validates our argument that a blind belief in data-driven models or direct application of a well-versed algorithm is not only counterproductive but risks interpreting the results with no attention to the dynamics of the system under study.

To encourage future extensions to *CHI* and future study in configuration subject areas, we share the full data-set used in Table 5, the python implementation of *CHI*, MARS & LARS, and the full set of results at https://www.kkant.net/config_traces/CHIproject.

7 CURRENT STATE OF THE ART AND CHALLENGES

7.1 *CHI* vs. State of Art.

Many domain specific articles speak about challenges pertaining to the configuration (or resource allocation) of networks [8, 9, 18], compute units or storage [3], operating systems [52], applications [37, 46], Cloud [25, 27], etc. A prominent approach in the literature on configuration settings has been the *performance influence model (PIM)* that captures the relationship between CVs and the performance [5, 28, 37]. PIM is almost entirely dependent on model training using available performance data and does not reflect or exploit any domain knowledge concerning either the relationships or the limitations that go beyond the range of training data. PIM like approaches look at the statistical influence of the configuration values and do not consider the design and architecture but are learned from observations [54]. A pure statistical model simply fits the data to a model but does not provide any insights into whether or why the real behavior is compatible with the statistical observations. In contrast, *CHI* aims at identifying the dominant properties of the CVs and quantifying their parameters using the data.

Xu et al. [49] report that the Apache server has more than 550 parameters and many of these parameters have dependencies and correlations, which further worsens the situation. Reference [28, 37] narrows this down to only nine CVs configuration options[12], but no rationale is given. In particular, the thread-pool size of Apache Server is not considered but is reported to be critical in Wang et al. [46]. *We believe that such issues can substantially benefit by exploiting the domain knowledge of the administrators instead of simply depending on the data, which could be misleading or inadequate.*

Probability based approaches for finding optimum configurations such as ConEx [21] are a variation of the PIM model that probabilistically sample the configuration space and then generate a machine learning (ML) model to predict an outcome (usually performance). However, the contribution of individual configuration parameters on the outcome is not modeled. Variability aware models proposed by Guo et al. [13] work on boolean CVs (being set true/false), but it is well known that arbitrary Boolean functions of this form simply cannot be learned [54]. As discussed by Zhang et al. [54], we show that performance functions are not arbitrary, but rather structured, hence can be potentially learned effectively.

In Ref. [2], authors build performance models for various applications to accurately distinguish the best or close-to-the-best configuration from the rest with only a few test runs. Using FLASH [30], authors sort or rate the configurations in order of the performance achieved. Ernest [45] design an ability to predict the performance of applications under various resource configurations to automatically choose the optimal configuration. In Ref. [24], authors study performance variability and answer how many repetitions of an experiment are likely to be needed to achieve high confidence in the results within a sufficiently narrow confidence interval. It is true that the performance could vary substantially due to variations in the workload; however, this is not a configuration issue. For our purposes, we are interested in average performance supported by a configuration, and not instantaneous performance or performance for specific workload inputs. The variability will make performance non-monotonic and multi-modal, which is precisely the characteristics that *CHI* uses.

[12]See http://tiny.cc/3wpwly

Velez et al.[44] & Ha et al. [14] observed that the influence of configuration parameters on performance is highly variable, i.e., some options are highly influential while others have little or no impact on the performance. Such performance variations have made it very challenging to predict the performance of an application running in the Cloud environment. *CHI* postulates the influence of the CVs as a convex-concave function (e.g., monotonic with diminishing returns) and thus quantifies the behavior of the configuration more descriptively.

Xu et al.[50] in their study of various application configurations reveal that about 4.7%–38.6% of the critically important CVs do not have any early checks and thereby cause a severe impact on the system's behavior. Xu focuses the study on CVs related to the system's Reliability, Availability, and Serviceability (RAS). This concept is in line with our approach in that *CHI* eliminates unimportant CVs (explained later as L_{un}) and expresses the service behavior with a measurable health index metric. Wang et al. [46] show that liberal allocation of a CV (i.e DB pool size) can lead to performance degradation. Further, their study shows the importance of considering the practical factors such as non-linear effect of resource allocation. *CHI* supports this observation and models the non-linearities in CVs as given above.

In Cloud environments, Zaman et al. [53] show that VM provisioning depends heavily on resource allocation which in turn affects economics and bidding process (e.g. VM_1 with 1x2-GHz CPU, 8GB memory, 1TB disk vs. VM_2 with 2x2-GHz CPU, 16GB memory, 2TB disk;). Zhu et al. [55] demonstrate the difficulty and infeasibility of the configuration tuning problem using common machine learning model-based methods. Wei et al. [47] and Moradi et al. [27] highlight the complexity of allocating multiple resource types in a study of heterogeneous resource allocation in Cloud VMs. Using practical data from a Cloud Storage environment [38], we show that *CHI* can help understand the effect of resource allocation. By understanding how multiple resource types (e.g., number of CPU cores, disk size, etc.) affect the performance/workload, *CHI* can aid users in reducing the monetary costs by choosing the right heterogeneous and economical resource allocation, thus be also cost-efficient.

It is also worth noting that although there are many Configuration Management Tools (e.g., CFEngine, Puppet, Ansible, etc. [31]), their job is only the application of provided settings to multiple resources consistently and ensuring that certain given relationships hold.

8 CONCLUSIONS

The behavior of all cyber systems depends on a set of configuration variables (CVs) which if set improperly could result in a variety of problems including sub-optimal performance. In this paper, we present a performance related *Configuration Health Index (CHI)* framework that can *quantify* the contribution of individual CVs towards the overall performance of the service. We evaluate *CHI* using a model-driven approach that exploits both the domain knowledge and the available data. We demonstrate the applicability of *CHI* using data sets from state-of-art systems and our study of Cloud Storage Gateway. In all cases, we demonstrate that *CHI* can learn the influence of CVs on service performance and accurately predict the behavior for new (yet unseen) configuration settings. We show that our approach works better than a pure data-driven characterization and can give a better *a priori insight* into the influence of the CVs on the system performance. We believe that *CHI* provides a substantial improvement over the state of the art and can be broadly applicable to a wide range of configuration management problems. We also demonstrate the dangers of the pure data driven models in that they might predict a wrong trend or eliminate important configuration variables. An approach that uses data judiciously along with the domain knowledge based constraints can address this problem.

ACKNOWLEDGEMENTS

We appreciate the support and active participation of Girisha Shankar (Ph.D student) from Indian Institute of Science, Bengaluru, India and Dr. Slobodan Vucetic of Temple University. The discussions with them were highly valuable in devising the solution and added to the techniques presented in the paper.

REFERENCES

[1] Alexander Pucher. Cloud Traces and Production Workloads for Your Research, 2020.

[2] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 469–482.

[3] Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A. C. Hippodrome: Running circles around storage administration. In *FAST* (2002), vol. 2, pp. 175–188.

[4] Bauer, L., Garriss, S., and Reiter, M. K. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Inf. Syst. Secur. 14*, 1 (June 2011), 2:1–2:28.

[5] Calotoiu, A., Beckinsale, D., Earl, C. W., Hoefler, T., Karlin, I., Schulz, M., and Wolf, F. Fast multi-parameter performance modeling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)* (2016), pp. 172–181.

[6] Daubechies, I., DeVore, R., Foucart, S., Hanin, B., and Petrova, G. Nonlinear approximation and (deep) relu networks. *arXiv preprint arXiv:1905.02199* (2019).

[7] Efron, B., Hastie, T., Johnstone, I., Tibshirani, R., et al. Least angle regression. *Annals of statistics 32*, 2 (2004), 407–499.

[8] Fernandes, G., Rodrigues, J. J., Carvalho, L. F., Al-Muhtadi, J. F., and Proença, M. L. A comprehensive survey on network anomaly detection. *Telecommunication Systems 70*, 3 (2019), 447–489.

[9] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 469–483.

[10] Forum of Incident Response and Security Teams. Common Vulnerability Scoring System. https://www.first.org/cvss/, 2017.

[11] Friedman, J. H. Multivariate adaptive regression splines. *The annals of statistics* (1991), 1–67.

[12] Guo, J., Chang, Z., Wang, S., Ding, H., Feng, Y., Mao, L., and Bao, Y. 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)* (2019), IEEE, pp. 1–10.

[13] Guo, J., Czarnecki, K., Apel, S., Siegmund, N., and Wasowski, A. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), IEEE, pp. 301–311.

[14] Ha, H., and Zhang, H. Deepperf: performance prediction for configurable software with deep sparse neural network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 1095–1106.

[15] He, S., Manns, G., Saunders, J., Wang, W., Pollock, L., and Soffa, M. L. A statistics-based performance testing methodology for cloud applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2019), ESEC/FSE 2019, Association for Computing Machinery, p. 188–199.

[16] Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., and Epema, D. H. The grid workloads archive. *Future Generation Computer Systems 24*, 7 (2008), 672–686.

[17] Iosup, A., Yigitbasi, N., and Epema, D. On the performance variability of production cloud services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2011), IEEE, pp. 104–113.

[18] Kakarla, S. K. R., Tang, A., Beckett, R., Jayaraman, K., Millstein, T., Tamir, Y., and Varghese, G. Finding network misconfigurations by automatic template inference. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020), pp. 999–1013.

[19] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[20] Karimireddy, S. P., Koloskova, A., Stich, S. U., and Jaggi, M. Efficient greedy coordinate descent for composite problems. In *The 22nd International Conference on Artificial Intelligence and Statistics* (2019), PMLR, pp. 2887–2896.

[21] Krishna, R., Tang, C., Sullivan, K., and Ray, B. Conex: Efficient exploration of big-data system configurations for better performance. *IEEE Trans. on Software Eng.* (2020).

[22] Lu, H., Freund, R., and Mirrokni, V. Accelerating greedy coordinate descent methods. In *International Conference on Machine Learning* (2018), PMLR, pp. 3257–3266.

[23] Makrani, H. M., Sayadi, H., Nazari, N., Dinakarrao, S. M. P., Sasan, A., Mohsenin, T., Rafatirad, S., and Homayoun, H. Adaptive performance modeling of data-intensive workloads for resource provisioning in virtualized environment. *ACM Trans. Model. Perform. Eval. Comput. Syst. 5*, 4 (Mar. 2021).

[24] Maricq, A., Duplyakin, D., Jimenez, I., Maltzahn, C., Stutsman, R., and Ricci, R. Taming performance variability. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 409–425.

[25] Masanet, E., Shehabi, A., Lei, N., Smith, S., and Koomey, J. Recalibrating global data center energy-use estimates. *Science 367*, 6481 (2020), 984–986.

[26] Milborrow, S., Hastie, T., and Tibshirani, R. Earth: multivariate adaptive regression spline models, 2014.

[27] Moradi, H., Wang, W., and Zhu, D. Online performance modeling and prediction for single-vm applications in multi-tenant clouds. *IEEE Transactions on Cloud Computing* (2021).

[28] Nair, V., Menzies, T., Siegmund, N., and Apel, S. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), pp. 257–267.

[29] Nair, V., Menzies, T., Siegmund, N., and Apel, S. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering 25*, 2 (2018), 247–277.

[30] Nair, V., Yu, Z., Menzies, T., Siegmund, N., and Apel, S. Finding faster configurations using flash. *IEEE Transactions on Software Engineering 46*, 7 (2018), 794–811.

[31] Önnberg, F. Software configuration management: A comparison of chef, cfengine and puppet, 2012.

[32] Papadopoulos, A. V., Ali-Eldin, A., Arzén, K.-E., Tordsson, J., and Elmroth, E. Peas: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 1*, 4 (2016), 1–31.

[33] Pereira, J. A., Martin, H., Acher, M., Jézéquel, J.-M., Botterweck, G., and Ventresque, A. Learning software configuration spaces: A systematic literature review. *arXiv preprint arXiv:1906.03018* (2019).

[34] Plan, Y., and Vershynin, R. The generalized lasso with non-linear observations. *IEEE Transactions on Information Theory 62*, 3 (2016), 1528–1537.

[35] Reiss, C., Wilkes, J., and Hellerstein, J. L. Google cluster-usage traces: format+ schema. *Google Inc., White Paper* (2011), 1–14.

[36] Shen, S., Van Beek, V., and Iosup, A. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2015), IEEE, pp. 465–474.

[37] Siegmund, N., Grebhahn, A., Apel, S., and Kastner, C. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 284–294.

[38] Sondur, S., and Kant, K. Towards automated configuration of cloud storage gateways: A data driven approach. In *International Conference on Cloud Computing* (2019), Springer, pp. 192–207.

[39] Sondur, S., Kant, K., Vucetic, S., and Byers, B. Storage on the edge: Evaluating cloud backed edge storage in cyberphysical systems. In *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)* (2019), pp. 362–370.

[40] Sondur, S., Shankar, G., and Kant, K. CHeSS: A Configuration Health Scoring System and Its Application to Network Devices. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)* (2020), pp. 250–257.

[41] Tateishi, S., Matsui, H., and Konishi, S. Nonlinear regression modeling via the lasso-type regularization. *Journal of statistical planning and inference 140*, 5 (2010), 1125–1134.

[42] Theisen, C., Herzig, K., Morrison, P., Murphy, B., and Williams, L. Approximating attack surfaces with stack traces. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 2, IEEE, pp. 199–208.

[43] Tibshirani, R. J., et al. The lasso problem and uniqueness. *Electronic Journal of statistics 7* (2013), 1456–1490.

[44] Velez, M., Jamshidi, P., Sattler, F., Siegmund, N., Apel, S., and Kästner, C. Configcrusher: towards white-box performance analysis for configurable systems. *Automated Software Engineering* (2020), 1–36.

[45] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)* (2016), pp. 363–378.

[46] Wang, Q., Zhang, S., Kanemasa, Y., Pu, C., Palanisamy, B., Harada, L., and Kawaba, M. Optimizing n-tier application scalability in the cloud: A study of soft resource allocation. *ACM Trans. Model. Perform. Eval. Comput. Syst. 4*, 2 (June 2019).

[47] Wei, L., Foh, C. H., He, B., and Cai, J. Towards efficient resource allocation for heterogeneous workloads in iaas clouds. *IEEE Transactions on Cloud Computing 6*, 1 (2015), 264–275.

[48] Westermann, D., Happe, J., Krebs, R., and Farahbod, R. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (2012), pp. 190–199.

[49] Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., and Talwadker, R. 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* (2015), pp. 307–319.

[50] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., and Pasupathy, S. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 619–634.

[51] Xu, T., and Zhou, Y. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys (CSUR) 47*, 4 (2015), 70.

[52] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[53] Zaman, S., and Grosu, D. A combinatorial auction-based mechanism for dynamic vm provisioning and allocation in clouds. *IEEE Transactions on Cloud Computing 1*, 2 (2013), 129–141.

[54] Zhang, Y., Guo, J., Blais, E., and Czarnecki, K. Performance prediction of configurable software systems by fourier learning (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), IEEE, pp. 365–373.

[55] Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., and Yang, Y. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 338–350.