Multi-Intention Aware Configuration Selection for Performance Tuning

Haochen He
hehaochen13@nudt.edu.cn
National University of Defense Technology, China

Zhouyang Jia*
jiazhouyang@nudt.edu.cn
National University of Defense Technology, China

Shanshan Li†
shanshanli@nudt.edu.cn
National University of Defense Technology, China

Yue Yu†
yuyue@nudt.edu.cn
National University of Defense Technology, China

Chenglong Zhou
zhouchenglong15@nudt.edu.cn
National University of Defense Technology, China

Qing Liao
liaoqing@hit.edu.cn
Harbin Institute of Technology, Shenzhen, China

Ji Wang
wj@nudt.edu.cn
National University of Defense Technology, China

Xiangke Liao
xkliao@nudt.edu.cn
National University of Defense Technology, China

ABSTRACT

Configuration tuning can improve software performance. Pre-selecting performance-related parameters can significantly reduce the search space during tuning. These works, however, are both limited by the specific workloads chosen to train their models. More importantly, they are unaware of user intentions other than performance but are also important (e.g., reliability, security). Given these limitations, we find that the configuration document often (even if it does not always), contains rich information about the parameters' relationship with many user intentions. However, documents might also be long and domain specific. Thus, we focus on guiding users in selecting performance-related parameters while warning about side-effects on non-performance intentions via mining documents.

In this paper, we first conduct a comprehensive study on 13 representative software containing 7,325 configuration parameters, and derive six types of ways in which configuration parameters may affect non-performance intentions. Guided by this study, we design SafeTune, a workload-independent method that pre-selects important performance-related parameters and warns about their side-effects on non-performance intentions. Evaluation on target software shows that SafeTune correctly identifies 6-22 performance-related parameters that are missed by state-of-the-art tools but have significant performance impacts (up to 14.7x). Furthermore, our case study demonstrates that SafeTune can successfully help the existing auto-tuner to warn about eight critical side-effects, such as data corruption.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Performance tuning, User intention, Software documentation

ACM Reference Format:
Haochen He, Zhouyang Jia, Shanshan Li, Yue Yu , Chenglong Zhou, Qing Liao, Ji Wang, and Xiangke Liao. 2021. Multi-Intention Aware Configuration Selection for Performance Tuning. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Configuration can change software behavior and thus enable customization to meet different user intentions. Among many possible user intentions, improving performance is one of the most common purposes. While modern software systems are often equipped with a large number of configuration parameters (e.g., HDFS [42] has 560 parameters, GCC [1] has 1,353). To address this problem, many existing works have applied various techniques to perform automatic configuration tuning [16, 17, 19, 24, 32, 37–39, 45, 47, 54]. However, these auto-tuners have two major challenges. First, they are extremely limited by the large number of parameters (the huge search space) [51, 54]. Second, they only consider performance improvement while being unaware of user intentions other than performance (e.g., reliability, security and functionality). However, one configuration parameter may impact multiple user intentions at the same time. For example, using a parameter to sacrifice reliability to gain performance is a common practice applied by software [3]. But users of safety-critical systems (e.g., industrial control system) may also intend to keep their systems reliable while improving

*Co-first author
†Corresponding author

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-1111-2222-3/44/55…$15.00
https://doi.org/10.1145/1122445.1122456

`innodb_flush_log_at_trx_commit`:
*"Controls the balance between strict ACID compliance for commit operations and higher performance... The default setting of 1 is required for full ACID compliance. Logs are written and flushed to disk at each transaction commit. With a setting of 0, logs are written and flushed to disk once per second. You can achieve **better performance** by changing the default value but then you can **lose transactions** in a crash."*

Figure 1: Configuration documents of MySQL

performance. In such cases, the parameter that affects the corresponding intentions should not be tuned.

Many recent works have been proposed to reduce the search space for those auto-tuners by *pre-selecting important parameters*. These works run dynamic performance experiments to measure performance changes that occur following changes of parameters, and use statistical [23] or machine-learning [34] methods to pre-select parameters that have significant impacts on performance. Though working in some scenarios, these methods still have two limitations: 1) The effectiveness of dynamic methods depends strongly on both workloads and environment, which often differ by cases. Moreover, these methods are difficult to deploy due to the expensive performance experiments. 2) These black-box methods **still focus only** on the performance impact of the parameters, but do not take impacts on other user intentions into account. Therefore, a tool that helps auto-tuners to pre-select important performance-related parameters in general, particularly one that pays strong attention to the *side-effects on other user intentions*, is needed.

In this paper, we propose SafeTune, a workload-independent and lightweight approach that provides *tuning guidance*, including identifying performance-related configuration parameters and warning about potential side-effects on non-performance user intentions to change the parameters. The key insight is that configuration parameters are often well-documented, especially in widely used software. SafeTune leverages configuration documents to understand the relationship between performance and other user intentions. For example, in Fig. 1, the document of parameter `innodb_flush_log_at_trx_commit` explains if and how the parameter affects performance in general, and warns about the potential side-effect (lose transaction) on the intention of reliability.

There are three main challenges in SafeTune. First, how configuration parameters affect performance and cause side-effects on non-performance intentions are unknown. Second, building a model to learn information from documents (which are written in natural language) requires large-scale training data, but there is no such public dataset exists. Third, the side-effect information is usually described implicitly, and documents can be very long and domain-specific. For the example shown in Fig. 1, expert knowledge is required to understand that "lose transaction" means hurting reliability in this context.

Therefore, we first conduct an empirical study to comprehensively understand how the configuration parameters can affect performance and cause side-effects on non-performance intentions such as reliability. To determine the general result from the study, we choose 13 software from four categories, including 7,325 parameters, as shown in Table 1. From this study, we obtain three heuristics to precisely filter out parameters unrelated to performance, and derive a categorization that contains six types of ways in which parameters can cause side-effects on non-performance intentions.

Based on these findings, SafeTune predicts the tuning guidance of a parameter for the given configuration document of the parameter. In light of the last two challenges, SafeTune takes two major steps: 1) We design a semi-supervised data expansion approach, which automatically expands the manually labeled training dataset. We manually assign the type of side-effect for parameters in a small-scale training data set during the study. Next, SafeTune uses association rule mining techniques in a progressive manner to mine natural language patterns from the dataset, then uses the patterns to enlarge this dataset with high precision. This step is necessary because manually inspecting all parameters is extremely expensive. 2) SafeTune trains a learning-based hierarchical model to capture important information from the document based on the expanded dataset and provide the final performance tuning guidance.

We evaluate SafeTune on software that has neither previously appeared in our study nor in the model training set. The results show that SafeTune can accurately identify performance-related parameters and their side-effects, scoring 81.3-85.1% in precision and 67.6-67.7% in recall. Compared with the state-of-the-art preselecting method [34], SafeTune can correctly find 117 performance-related parameters that are missed by the method, and some of the these parameters have huge (up to 14.6x) performance impacts. Moreover, SafeTune correctly covers 29/32 performance-related parameters that identified by the method. The false positive rate is 12.7%. Furthermore, we conduct a case study in which we apply SafeTune with a state-of-art and popular auto-tuner, OtterTune [43], which has 1.1k `GitHub` stars. The results show that SafeTune can help to prevent eight side-effects (covering four types) caused by OtterTune that lead to severe consequences.

Our main contributions can be summarized as follows:

- We conclude six types of ways in which performance-related parameters can affect non-performance user intentions from an empirical study of 13 open-source software.
- We design and implement a workload-independent and lightweight approach, SafeTune, to identify performance-related parameters and their side-effects. All data (including more than 7,325 annotated parameters) and source code can be found in our public repository:

  https://github.com/Anonymous-user-2021/SafeTune
- We evaluate SafeTune on the target software. The results show that SafeTune finds 6-22 performance-related parameters that have large performance impacts (up to 14.7x) but are missed by state-of-the-art works. Furthermore, SafeTune helps the state-of-the-art auto-tuner prevent eight critical side-effects on other user intentions.

2 UNDERSTANDING PERFORMANCE-RELATED CONFIGURATION

To comprehensively understand which configuration parameters affect software performance, along with what side-effects those performance-related parameters may have on non-performance intentions, we conduct an empirical study on 7,325 parameters from 13 open-source software systems. From this study, we first derive

Table 1: Studied Software and Configuration Parameters

Category	Software	Popularity‡	# Params†
Database	MySQL	6.8k	919
	Cassandra	6.8k	116
	MariaDB	3.9k	274
Web Service	Apache Httpd	2.7k	571
	Nginx	14.5k	710
Distributed Service	Hadoop Common	11.8k	313
	MapReduce	11.8k	198
	Apache Flink	16.8k	441
	HDFS	11.8k	560
	Keystone	4.4k	394
	Nova	2.7k	844
Developer Tool	GCC	5.3k	1,335
	Clang	2.8k	650
Total			7,325

†The number of configuration parameters. ‡ Github stars

three heuristic strategies to help filter out parameters unrelated to performance. We then conclude six different types of ways in which those performance-related parameters may cause side-effects. These findings are used to guide the design of SafeTune.

2.1 Data Collection

We study the configuration documents of the software. The configuration document explains the detailed semantics and their relationships with user intentions (e.g., performance). It has two unique advantages. First, it provides a general but comprehensive understanding of configuration parameters that does not rely on specific workloads; then, it contains multiple user intentions (e.g., text in bold in the box shown in §1). We studied 13 open-source software systems from four different domains, as shown in Table 1. These four categories are chosen from the most popular products provided by famous cloud vendors [7, 9, 11] and are representative among highly-configurable software systems [28, 33, 43, 44, 52, 54]. Moreover, these software systems are: 1) usually located in server-side and accordingly have higher demands in terms of performance, reliability, etc; 2) mature and widely used, with at least 2.7k Github stars; 3) highly configurable (each has more than 100 configuration parameters) with well-maintained configuration documents. We collected configuration parameters and their documents from two main sources: the official websites and the configuration files (e.g. XML-based configuration files). Each of the collected data point is in the form of <*parameter name, description*>. We filter out parameters without description. Finally, we collected 7,325 from these software systems.

Identifying Performance-related Parameters. To understand the side-effects of performance-related parameters, we manually studied the documentation of configuration parameters. Studying all parameters is extremely expensive; accordingly, we conclude three heuristic strategies to filter out parameters that have no impact on the performance of software. 1) Parameters indicating the location of resources. Descriptions of these parameters contain phrases such as "path of", "port of", "address of", and "location of". These typically have little impact on performance. 2) Parameters marked as "unused" or "deprecated" in the documentation. 3) Parameters set for compatibility reasons. These parameters are usually designed to support old behavior in old versions of software. Parameters in this category are filtered by the keywords "version", "compatibility" and "legacy". We apply these heuristics to all 7,325 parameters and filter out 1,071 parameters. Note that these heuristics are set to quickly filter out some of parameters unrelated to performance so that the others still need to be further filtered out by SafeTune. We then randomly sample 1,292 (20%) of the 6,254 parameters to study. Two authors with at least three years of research experience in configuration independently identified each parameter as either performance-related or not according to its description. Once a disagreement occurred, a third author was involved until a consensus is reached. Finally, we obtained 525 performance-related parameters for further study.

Table 2: Side-effects on non-performance intentions

Type of Side-effects	# Params
Lower reliability	33
Lower security	46
Reduced functionality	138
Lower performance (other users)	68
Higher cost	114
Limited side-effect	126
Total	525

2.2 Side-effects on Non-performance Intentions

We study the 525 performance-related parameters to understand the side-effects they may cause. We follow the same manual cross-check methodology as the above section; additionally, to make the results more consistent, the rest of the authors randomly review a fifth of the parameters studied in weekly meetings. This process took 200 working hours and lasted for eight weeks. Finally, we conclude six different types of ways in which these performance-related parameters may cause side-effects on non-performance user intentions.

As a result, we find majority (76.0%, 399/525) of performance-related parameters have side-effects on non-performance intentions. These parameters can affect common user intentions, such as reliability, security and functionality. Table 2 shows the number of parameters falling into each types of side-effects. This result indicates a strong demand for tuning tools to warn about these side-effects for end users. Subsequently, we describe each type and the criteria used to classify it in our study.

Lower reliability. These parameters improve performance at the cost of decreasing the level of reliability protection. For example, `innodb_flush_log_at_trx_commit` controls the ACID level of MySQL, the value of "1" ensures strict data protection by executing *fsync* at every commit. Changing this value to "2" improves the performance by reducing the calls to *fsync* at the risk of losing data during a power loss. To avoid affecting user intention of reliability, tuning tools should warn users about the risk associate with improving performance. *Criteria to classify*: These parameters are usually related to the way data are persistent to the hard drive and policies to protect data from single-point failure, and they are

usually documented as "replication level", "update interval", "write to disk", etc.

Lower security. These parameters improve performance at the cost of a lower level of security protection. For example, the parameter `PrivilegesMode` in Apache Httpd controls the way requests are processed. The value `"SECURE"` means that all requests are run in a secure sub-process, but with more overhead. When changing to `"FAST"`, requests are run in-process, speeding up the software but opening up the chance for malicious attackers to utilize the in-process module to escalate privileges. To avoid affecting user intention regarding security, such parameters should be warned about. *Criteria to classify*: These parameters are usually related to common security policies such as encryption, authentication and privacy protection, and they are usually documented as "enable/disable authentication", "enable/disable SSL", "whether (a kind of data) should be encrypted", etc.

Reduced functionality. These parameters improve performance at the cost of reduced functionality. For example, the parameter `adl.feature.ownerandgroup.enableupn` controls whether an additional process should be performed to convert users and groups in FileStatus/AclStatus response to a user-friendly name. Disabling this function saves a large amount of computation (as documented: "*for optimal performance, false is recommended*"); however users intending to enable this function should be provided with warnings. *Criteria to classify*: These parameters are usually documented as "enable/disable (a feature)", "control output", "collect information of (a component)", etc.

Lower performance (other users). These parameters can improve performance only for specific workloads run by some users, and may hurt others run by other users. Taking the parameter `max_seeks_for_key` in MariaDB as an example, it controls the estimated maximum cost for look-ups on table's index. Decreasing the value makes MariaDB prefer index scan than full table scan. But the index scan is only faster than the full table scan when the cost for index look-ups is low in actual (i.e., low cardinality of the index), which is completely workload-dependent. If another user runs a different workload, the inappropriate value may cause the performance problem [2]. So these parameters should be tuned with caution. *Criteria to classify*: These parameters typically control the internal argument of a specific algorithm, data structure and model, and they are usually documented as "threshold of", "ratio of", "upper/lower bound of", etc.

Higher cost. These parameters improve performance at the cost of consuming more system resources (e.g. CPU cores, memory, bandwidth). For example, `dfs.image.parallel.threads` in HDFS sets the number of threads used to load the image. A higher value of this parameter results in higher parallelism and reduced loading time, while more CPU cores may be used. In Amazon Web Services [7], four more CPU cores for a 32GB memory instance can cost 72$ per month. In cases where a user's budget or hardware resources are limited, changing these parameters may still indirectly affect user intentions (e.g., unexpected bill charges). *Criteria to classify*: These parameters usually control the system resources allocated to the software, and they are usually documented as "size of buffer", "number of workers", etc.

Limited side-effects. These parameters improve performance with limited side-effect on non-performance intentions. First, some parameters improve performance by applying certain optimization strategies. For example, when `index_merge` in MySQL is turned on, MySQL can better utilize index and read from a single table rather than across multiple tables. Such optimization has a limited impact on the system, thereby causing limited side-effects on user intentions. It is recommended for tuning tools to tune such parameters in the first place. *Criteria to classify*: These parameters usually control optimization strategies such as caching, load balancing and compression, and they are usually documented as "enable (a kind of optimization)", "whether to (do optimization)", etc.

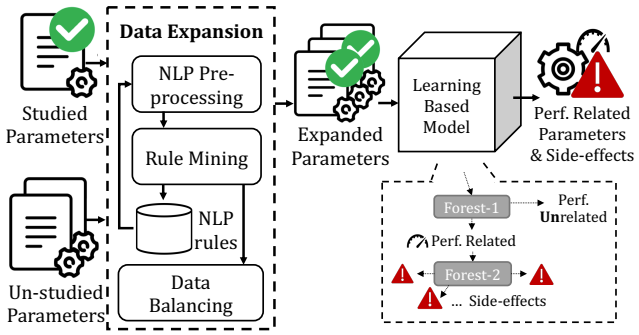


Figure 2: Overview of SafeTune.

3 SEMI-SUPERVISED DATASET EXPANSION

Given the side-effects summarized in the study, our goal is to build a model that can automatically identify the performance-related parameters and warn about their side-effects on non-performance intentions. Training a model to get information from natural language requires data at a large scale. We refer to the configuration parameters and their side-effect types (labels) as training data; hence, SafeTune has seven labels (including six side-effects and performance unrelated parameters). However, the training data obtained from the study is insufficient (i.e., does not exceed 100 for some types), and manual labeling of the unstudied data is extremely expensive.

During the empirical study process in §2, we find that descriptions of parameters of the same type share certain linguistic patterns. For example, descriptions of parameters that may lead to higher cost usually contain phrases like "size of buffer" or "number of threads". With these patterns, we will be able to enlarge the dataset with less manual effort. We therefore design a semi-supervised data expansion approach that utilizes natural language processing (NLP) and association rule mining (ARM) techniques to automatically mine the patterns (i.e., association rules) in a progressive manner in order to enlarge the amount of labeled data in the study. As shown in the left part of Fig. 2, the data expansion process contains three main steps: First, SafeTune uses NLP to pre-process the manually studied configuration documents and normalizes words to highlight the most informative words. Second, it mines the pre-processed documents to obtain a set of rules using ARM. With these rules, it matches configuration documents that are not involved in the study to increase the available training data. The above two steps proceed iteratively until no configuration documents can be enlarged by the rules. At the third step, SafeTune balances the data of each type of side-effect to avoid a biased training set.

Table 3: The rule examples of each side-effect types mined by SafeTune

Side-effect type	Rule example	Support	Confidence	Matched description
Lower reliability	(NOUN, write), (NOUN, level)	7	0.875	Sets the current transaction's synchronization† level.
Lower security	(VERB, check), (NOUN, security)	6	0.857	Sets how deeply mod_ssl should verify before deciding that the clients do not have a valid certificate.
Reduced functionality	(NOUN, level), (NOUN, information)	6	0.857	Verbosity of SQL debugging information: 0=None, 100=Everything.
Lower performance (other users)	(NOUN, time), (ADP, for), (NOUN, resource)	6	0.857	The time for which retry cache entries are retained.
Higher cost	(VERB, set), (NOUN, amount), (NOUN, resource)	12	0.923	This value controls the number of cache directives that the NameNode will send over the wire in response to ...
Limited side-effect	(VERB, enable), (NOUN, optimization)	12	0.857	Enables or disables genetic query optimization.

† words underlined are those matched by rules.

3.1 Pre-processing

The goal of pre-processing is to normalize words and highlight important information in the description of parameters. Pre-processing contains three steps: lemmatization, reduction, and substitution.

At the lemmatization step, since we are not interested in the grammar features contained in the documents, we transform each token to its original form to eliminate possible third-person or plural format effects; for instance, the word "specified" is transformed into "specify". In the reduction step, we remove the words that are not likely to convey useful information and retain the informative ones. During our study, we find that 4 kinds of words play an important role in deciding the possible side-effects. These are: 1) *noun*s, which directly point out the entities on which parameters will have an effect; 2) *verb*s, which are actions related to parameters and tend to directly lead to the impact; 3) *adverb*s/*adjective*s, which describe the effects of parameter value changes. We therefore extract the part-of-speech (POS) information of each word and retain works with POS of *noun*, *verb*, *adv*, and *adj*. In the substitution step, we replace special words to prevent the model from being distracted by unrelated information. For example, we replace the parameter names that appeared in the documentation with "CONFIG" and replace numbers with a fixed number. Moreover, to highlight the semantic knowledge in the documents, we replace words that appear in our manually collected synonyms list, which is obtained by referring to domain-specific resources [4, 5, 22]. For example, the words "commit", "update" and "sync" are replaced with "write", since they have similar meanings. We implement the pre-processing part using spaCy [30]. We provide an example to demonstrate the above process: the description *"Sets the current transaction's synchronization level"* will be converted to `[(VERB, set), (ADJ, current), (NOUN, transaction), (NOUN, write), (NOUN, level)]`.

3.2 Mining Association Rules

The goal of this step is to find the sub-sequences that appear exclusively and frequently in descriptions of the specific type of side-effect of performance-related parameters. These sub-sequences are association rules that distinguish different types of side-effect. For the pre-processed sentences obtained from § 3.1, we utilize the FEAT algorithm [25] to mine association rules for each type of side-effect. SafeTune utilizes it by adding the label to the end of the description and mining the most frequent sub-sequences (rules) co-occurring with the label. For example, a rule mined by the algorithm is `[(NOUN, write), (NOUN, level)]`, which is a sub-sequence of the example above. Also, SafeTune outputs the *support* and *confidence* of each association rule, and they are defined as follows:

$$support = |rules_i| \quad \text{where the } rule_i \text{ matches the document of the type of the } i^{\text{th}} \text{side-effect, and}$$

$$confidence = \frac{support}{|rules_i^*|} \quad \text{where the } rule_i \text{ matches the document of any type of the side-effect.}$$

The *support* of the $rule_i$ is defined as the number of occurrences of the rule matching i^{th} type given that the rule appears. Moreover, its *confidence* is the conditional probability that a document is the i^{th} type of side-effect when matching this rule. For example, the rule above has a *support* of 7 and *confidence* of 0.857, which represents a strong signal of its side-effect type (i.e. *Lower reliability*). Table 3 presents rule examples mined for each type.

In the mining stage, SafeTune may obtain millions of rules. As this set may contain rules with low quality, we retain only the rules that are sub-sequences of other rules with the same *support*, since shorter rules are more general and are thus more likely to expand more data. Moreover, this expanded training data will be directly used to train the SafeTune model. Thus, the soundness of the expanded training data is important. Furthermore, we do not expect the expansion process to expand all parameters to the training data (completeness). Therefore, we drop out the rules whose *confidence*s are lower than 0.85. As we will show in §5.2, with this level of *confidence*, the data expansion method can achieve an optimal balance between precision and recall.

3.3 Expanding Dataset Progressively

The goal of this step is to use rules mined from the studied data to match configuration documents to the greatest extent possible to enlarge the training data. For parameters not involved in the study, SafeTune pre-processes their documents as in §3.1. SafeTune then matches rules mined in the previous step to identify the candidate to be expanded. Note that one parameter document may match multiple rules of different types. SafeTune calculates the sum of matched rules' *confidence* of each type respectively and take the label with the highest score. If no rule is matched, the parameter will not be expanded. If the two steps described in §3.1 and §3.2 are applied only once, the data expanded may still be insufficient.

SafeTune expands the dataset in a progressive manner. The process terminates when there is no data remaining that can be expanded.

3.4 Balancing Data

The goal of this step is to make the training data balanced in order to avoid the model being biased towards the majority classes. After the data expansion via rules matching, the parameters have no impact on the performance account for 72.2% of the entire dataset, while the performance-related parameters that cause *Lower security* contain only 4.1%. This unbalance causes the model to be easily biased towards the performance-unrelated parameters. To avoid this bias, SafeTune over-samples the parameters in the minority types of side-effects to ensure that they are the same as the number of performance-unrelated parameters. Since the soundness of the expanded data is our main concern, we need to lower the probability of incorporating false data. Borderline-SMOTE [27] is a widely used over-sampling method for unbalanced data. Compared with other methods, it can effectively avoid generating samples from the "danger area" (i.e., samples near the borderline of different classes). After balancing the training data, SafeTune applies the pre-processing step as in § 3.1 to the data set to normalize words and highlight important information in the training set.

4 IDENTIFYING PERFORMANCE-RELATED PARAMETERS AND SIDE-EFFECTS

In this section, we introduce how SafeTune predicts the tuning guidance for configuration parameters. As shown in the right part of Fig. 2, SafeTune takes configuration documents that have been expanded and pre-processed from the data expansion step as input, then outputs tuning guidance including performance-related parameters and their side-effects on non-performance intentions.

Random forest (RF for short) [20] is an appropriate algorithm for the text classification task. It can precisely capture the difference between types of side-effects and is more robust than a single decision tree. It is also more lightweight and more interpretable than deep learning models like CNN. More importantly, neural networks usually demand millions of data for training, which is not accessible for configuration documents. The labels of input data in our task are hierarchical (i.e., level-1: performance-related/unrelated; level-2: six different side-effects only for performance-related parameters). Moreover, the classic RF algorithms do not account for hierarchical datasets. Inspired by [26], we build a two-level hierarchical random forest model, as shown in the bottom right part of Fig. 2. RF-1 is used to identify the performance related parameters from unrelated ones. All training data are used to train RF-1. Moreover, RF-2 is used to predict side-effects (among the six side-effects) for the performance-related parameters, while only the performance-related parameters in the training set are used to train RF-2. The label of the data to be predicted is decided by the multiplication of the probabilities given by the two RFs. For example, if the probabilities given by the two RFs of a configuration parameter are RF-1 : $[0.3, 0.7]$ and RF-2 : $[0.04, 0.2, 0.1, 0.5, 0.1, 0.06]$, then the parameter will be labeled as "performance-related" (since $0.7 > 0.3$), as well as the "*Higher cost*" label ($0.7 \cdot 0.5 = 0.35$, which is greater than all the others).

RF requires embedding of the input documents written in natural language. SafeTune embeds each parameter's document using TF-IDF [41], a widely-used method in information retrieval. For its part, TF-IDF treats each document as bag-of-words, ignoring the sequence information. In our task, "write the data" and "...data. Write the..." can express completely different meanings. Hence, SafeTune considers at most three consecutive words (i.e., unigram, bigram and trigram) when calculating the TF-IDF embedding. Note that the embedding may be sparse (e.g., 2,000 dimensions, with 1,990 zeros), and the RF algorithm only selects several dimensions each time to train a decision tree [20]. Thus, SafeTune applies principal component analysis (PCA) [13] for the TF-IDF embedding. We ensure that the PCA preserves 99% of the information from the initial TF-IDF embedding.

5 EVALUATION

We implement SafeTune with sklearn [21] and randomForest(R) [36]. All experiments are conducted on machines with a 48-core Intel-Xeon 2.2GHz processor, Tesla V100 GPU, 64GB RAM, and 1TB hard disk, with Ubuntu 18.04 LTS, and Python 3.6.8. We evaluate the effectiveness of SafeTune by answering the following questions:

- **RQ1: Accuracy of predicting tuning guidance and data expansion.** How accurate is SafeTune in identifying performance-related parameters and predicting their side-effects on non-performance intentions? How accurate is the automatically expanded data?
- **RQ2: Comparison between SafeTune and the state-of-the-art tool.** Can SafeTune cover performance-related parameters that identified by the existing tool? Can SafeTune identify performance-related parameters that are missed by the existing tool?
- **RQ3: Effectiveness of SafeTune on helping performance tuning.** Does the existing auto-tuner produces potential side-effects on other user intentions? Can SafeTune help the tool to prevent those side-effects? How severe are these side-effects?

5.1 RQ1: Accuracy of Predicting Tuning Guidance and Data Expansion

To answer RQ1, we evaluate the accuracy of SafeTune in identifying performance-related parameters and predicting their side-effects. To avoid over-fitting, we conduct experiments on software that are neither studied nor included in the training set. Since SafeTune automatically expands the training set, we also evaluate the accuracy of the expanded data.

5.1.1 Accuracy of Tuning Guidance. We evaluate SafeTune on PostgreSQL [40], Squid [46] and Apache Spark [50]. These software are not included in our study, but are also popular (at least 1k `GitHub` stars) and from different domains, written in different programming languages. Thus, we believe they can provide sufficient generality. The three software have 252, 266 and 217 parameters respectively. We follow the same methodology as in § 2.1 to manually label the parameters and use them as the test set. We use the 1,292 studied parameters as the initial training set. SafeTune then expands the training set and obtain 14,528 pieces of training data.

Table 4: Precision and recall in predicting performance-related parameters and their side-effects

Software	SafeTune				SafeTune$_{\text{w/o exp.}}$				SafeTune$_{\text{ideal}}$			
	PR		SE		PR		SE		PR		SE	
	preci.	recall	preci.	recall	preci.	recall	preci.	recall	preci.	recall	preci.	recall
PostgreSQL	0.873	0.764	0.812	0.629	0.713	0.545	0.623	0.481	0.873	0.777	0.847	0.685
Squid	0.872	0.602	0.830	0.623	0.693	0.426	0.589	0.439	0.891	0.632	0.868	0.652
Spark	0.801	0.669	0.792	0.651	0.611	0.532	0.510	0.389	0.855	0.662	0.820	0.648
Overall	0.851	0.677	0.813	0.676	0.498	0.577	0.553	0.439	0.881	0.691	0.847	0.662

PR: effectiveness of predicting performance related parameters. **SE**: effectiveness of predicting side-effects. *precis.*: precision. $_{\text{w/o exp.}}$: only use studied data to train SafeTune. $_{\text{ideal}}$: replacing labels of expanded data (may contain incorrect ones) in the training data set with manually checked labels.

Each item in this dataset is in the form of <*parameter name, description (embedded),* $label_{level\text{-}1}$, $label_{level\text{-}2}$>, where $label_{level\text{-}1}$ is one of the two values {"*Performance-related*", "*Performance-unrelated*"}, and $label_{level\text{-}2}$ is one of the following: {"*Lower reliability*", "*Lower security*", "*Reduced functionality*", "*Lower performance (other users)*", "*Higher cost*", "*Limited side-effects*"}. Last, SafeTune is trained by this dataset.

To measure the accuracy of predicting performance-related parameters (**PR** for short in Table 4), we use precision and recall. To assess the accuracy of predicting side-effects (**SE** for short in Table 4), we calculate the averaged precision and recall of each type (i.e., Micro-precision/recall [6] of six side-effects) to measure SafeTune as a whole. SafeTune applies data expansion to improve the accuracy. To evaluate the usefulness of this component, we remove it (using only the initial 1,292 parameters as the training set) and conduct experiments with identical test data to draw a comparison. This is denoted as **SafeTune**$_{\text{w/o exp.}}$ in Table 4.

Result and Analysis. First, SafeTune can identify performance-related parameters with a precision of 85.1% and a recall of 67.7%. The false negatives occur because many of them contain technical terms in documents that are difficult for SafeTune to understand. For example, `ssl_ecdh_curve` is documented as "*sets the curve to use for ECDH*", but this does not explain that "ECDH" is an agreement protocol that allows two parties to establish a shared secret, which affects performance. Worse yet, these technical terms rarely appear in the dataset. The false positives occur mainly because some expressions mislead SafeTune. For example, the document of `spark.eventLog.overwrite` is "*whether to over write any existing files*", but whether or not "overwrite" is performed does not affect performance in this case; however the word "write" misleads SafeTune into identifying the parameter as related with persisting data to disk, thereby affecting performance. Moreover, technical terms are one of the main causes of false positives.

Table 4 demonstrates the results of predicting side-effects on the non-performance intentions of these performance-related parameters. Overall, SafeTune can reach a precision of 81.3% and a recall of 67.6%. These false positives occur because many parameters have very complex logic described in the long document, making it challenging even for human to identify the type of side-effect. Another reason is that some expert knowledge cannot be precisely captured by SafeTune. For example, `wal_writer_flush_after` controls "amount of WAL written out by WAL writer that triggers a flush"; this parameter is falsely identified as *lower reliability*, but it actually controls the frequency of WAL flush, and the optimal frequency is workload-dependent (side-effect: *lower performance for other user*). The expert knowledge is that the WAL loss during power loss would not corrupt data (not *lower reliability*). The false negatives are mainly because the descriptions may miss context. For example, `cpu_tuple_cost` is described as "*sets the planner's estimate of the cost of processing each row during a query*", but the context of this description is that the "planner" is a component that will choose the quickest query plan, and the "cost" is a workload-dependent argument of the choosing algorithm. Without such context, SafeTune fails to identify it with the side-effect of *lower performance (other user)*.

The 6^{th} - 9^{th} columns of Table 4 show the results after removing the data expansion component from SafeTune. Without the data expansion, SafeTune cannot fully learn features in parameter documents, the precision drop by 26.0-35.3% and the recall drop by 10.0-23.7% respectively. Therefore, the data expansion is essential for SafeTune. *In conclusion, the result indicates SafeTune can achieve good precision and acceptable recall in pre-selecting performance-related parameters and predicting side-effects.*

5.1.2 Accuracy of Data Expansion. SafeTune uses data expansion to enlarge training data. So the quality of enlarged data may affect the effectiveness of final tuning guidance provided by SafeTune. Therefore, we also evaluate the correctness of the data expanded by the expansion method. To achieve this, we need the ground truth of these data. So we manually label all the 7,325 parameters and cross-checked them in the same way as described in § 2.2. This process took 400 working hours and last 10 weeks. We make all these data publicly available in the repository. Note that this manual work is only needed in this paper to evaluate SafeTune, but not for users of SafeTune. We use precision to measure the correctness of the data expanded by the expansion step, and expansion rate to measure the rate of the data that can be correctly expanded to all the unlabeled data (100% expansion rate means all the unlabeled data can be correctly expanded). Also, we use Micro-precision [6] and averaged expansion rate to measure the overall result. Further, we evaluate the impact of the incorrectly-expanded data on SafeTune. So we replace the labels of expanded data in the training set in § 5.1 with manually annotated labels (ground truth), and keep other experiment settings the same. We denote the model trained by this data set as **SafeTune**$_{\text{ideal}}$ in Table 4.

As described in § 3, SafeTune only keeps rules whose confidences are higher than a threshold to improve precision. Therefore, we evaluate the influence of different thresholds (from 0.10 to 0.95, a step of 0.05) on the precision and recall of the expanded data.

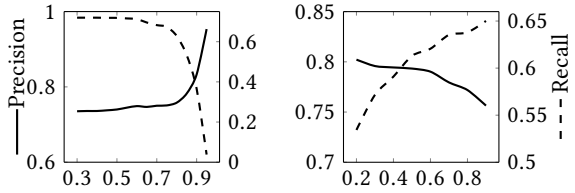
Table 5: Precision and expansion rate of the expanded data

Data Label	Precision	Expansion rate
Lower reliability	0.847	0.393
Lower security	0.808	0.372
Reduced functionality	0.832	0.585
Lower performance (other users)	0.819	0.552
Consuming more resource	0.910	0.670
Limited side-effect	0.801	0.464
Performance-unrelated	0.931	0.702
Overall	**0.864**	**0.594**

Another interesting question is how much studied (labeled) data do we need to conduct the expansion? Obviously, it is less useful if the expansion approach needs a majority of studied data and can only expand the rest minority. Therefore, we evaluate the influence of proportion on the precision and recall of the expanded data. Note that in our evaluation, all data are manually labeled to provide the ground truth, so we can simulate any proportion of studied data against those need to be expanded. For each sampling proportion p, we apply the approach in § 3 to expand the rest $1-p$ data. This process is repeated 10 times to eliminate the occasionality caused by the random sampling. We use Micro-precision and Micro-recall of the $1-p$ expanded data.

Result and Analysis. Table 5 shows the precision and recall of the expanded data of each type of side-effects. SafeTune performs well in expanding the performance-unrelated parameters and those may consume more resource. For the rest types like *Lower reliability* and *Lower security*, the number of parameters in these types are fewer in the initially studied data set and thus do not have many distinct features compared with other types. The result of the impact on SafeTune of those incorrectly-expanded data is shown in the last 4 columns in Table 4. The incorrectly-expanded data only cause about 3.0-3.4% degradation to precision (comparing the **SafeTune** with **SafeTune**$_{\text{ideal}}$ series), but without the data expansion approach, the precision will reduce significantly . And with the expansion, we can reduce about 63.0% expensive manual effort. This significantly outweighs the drawback of the incorrectly expanded data.

The result of choosing a proper threshold for rule confidence and proportion of studied data is shown in Fig. 3. First, Fig. 3a shows the Micro-precision/recall changes against different threshold values. As expected, the precision increases as the confidence threshold of the mined rules increases, and the recall is the opposite. As described in § 3, SafeTune honors precision rather than recall. When the threshold increase from 0.85 to 0.9, the precision increases by 5.1% while suffering a recall drop of 16.8%, which means about 700 parameters can not be automatically expanded but the precision improvement is small. Therefore, we set the confidence threshold as 0.85 in SafeTune. Fig. 3b shows the Micro-precision and Micro-recall of different proportions p of studied data. It is shown that, generally, with more data studied, the precision of the mined rules will slightly drop while the recall will increase. This is because that with more studied data, the variety increases accordingly, more rules can be generated and more unlabeled parameters can be expanded. This improves the recall but becomes more likely to make mistakes. Also, SafeTune honors more precision than recall. So, we use $p = 0.2$.



(a) Confidence threshold **(b) Proportion(p) of studied data**

Figure 3: The influence of confidence threshold and proportion of studied data on the precision and recall of the expanded data.

5.2 RQ2: Comparing with State-of-the-art Tool

We compare SafeTune with [34], a state-of-the-art tool to select important parameters by running performance experiments and choosing parameters that leads to significant performance change via machine learning techniques. This prior work [34] opens their results which include the Top-n important parameters (32 in total) in PostgreSQL and Cassandra predicted by them. So we evaluate SafeTune in these two software to compare with the prior work [34]. The two software has 252 and 117 configuration parameters respectively. We train SafeTune on the same data set (does not include PostgreSQL) as in § 5.1 but excluding Cassandra. To get the ground truth of performance related parameters in the two software, we manually label the parameters in the same way as 2.1. Note that this manual work is only for the evaluation, but is not needed for users of SafeTune.

Note that the prior work [34] predicts the performance related parameters by concrete performance experiments. While SafeTune is based on configuration documents. To prove that the performance related parameters predicted by SafeTune do have impact on performance, we run performance tests under different values of parameters. And we measure the performance impact by the factor of performance change before and after the parameter's value change. We collect the performance tests from popular benchmarks including: TPC-C [14], TPC-H [15] and ca-stress [8]. Each test is repeatedly run 10 times to get stable results. We refer to the performance as the 99%/100% latency (tail latency), mean throughput and single query execution time.

Result and Analysis. Experiment results show that SafeTune can identify 117 performance related parameters **that are missed by** the prior work [34]. Among the 117 parameters, 28 show significant performance impacts (up to 14.6x) in our performance testing. Table 6 shows the Top-3 parameters that have big performance impact for each software. For example, changing `enable_sort` from 1 to 0 makes the execution time of TPC-H.q17 degrades from 19.1 seconds to 279 seconds (14.6x). However, this parameter even never appears in the rank-list of the prior work [34]. This happens because `enable_sort` only affect queries that both contain `GROUP BY` and `ORDER BY` operations, but the workload used by the prior work [34] does not contains that. Another interesting thing is, the top-ranked parameter `fsync` given by the prior work [34] only has 1.7x performance impact and gets rank-12 during our testing. The reason is similar with above. Note-worthily, SafeTune gets this result without any heavy performance experiments (the prior work [34] consumes 3,750 machine hours). The performance testing in our

Table 6: Part of performance related parameters missed by state-of-the-art tool (full result in public repository).

Parameter Missed	Workload	Chg.	Metric
PG.enable_sort	TPC-H.q17	**14.6**x	Exe. Time†
PG.enable_nestloop	TPC-H.q2	**6.4**x	Exe. Time†
PG.enable_indexscan	TPC-H.q13	**3.6**x	Exe. Time
CA.native_transport_max_concurrent	ca-stress.w	**13.5**x	Tail Lat.
CA.hinted_handoff_throttle_in_kb	ca-stress.w	**6.7**x	Tail Lat.‡
CA.allocate_tokens_for_keyspace	ca-stress.rw	**1.2**x	Throughput

Chg.: Performance change before and after parameter change; **PG**: PostgreSQL; **CA**: Cassandra; †Query Execution Time; ‡Tail Latency.

evaluation cost about 700 machine hours while the time consumed by SafeTune is only 2 hours for the one-time-effort training step and less than 10 seconds for the prediction step.

SafeTune can successfully cover 29 out of 32 (90.6%) parameters given by the prior work [34]. In total, SafeTune produces 17 false positives (precision: 87.3%), and misses 45 out of 191 parameters (recall: 76.4%) that are manually confirmed from the documents to be performance related. Most of the false positives are parameters that have long description to explain the domain knowledge behind the parameter, rather than what turning on/off the parameter will affect. And the explanations may contain phrases that appear frequently in performance related parameters, thereby misleading SafeTune. For example, turning on `zero_damaged_pages` only reports a warning (performance unrelated), but its description explain a lot why this warning happens. The 3 cases that the prior work [34] identifies but missed by SafeTune are: 1) `commitlog_segment_size_in_mb` which controls commitlog file segments, needing strong domain knowledge to understand but there are rare similar cases in the training set. 2) `compaction_throughput_mb_per_sec`, whose description is too brief to be understood by SafeTune. 3) `default_statistics_target` whose description contains too much performance unrelated explanations that distract SafeTune.

Note that both the prior work [34] and our evaluation are limited by the workload. Some parameters do not trigger big performance change under the selected workload. For example, `max_logical_replication_workers` controls maximum workers a logical replication transaction can use. This parameter affects performance only when PostgreSQL is in a replication process, but this workload does not included in any of selected benchmarks. In the evaluation, we use richer types of workloads so we observe more performance related parameter than the prior work [34].

In conclusion, SafeTune can identify many performance related parameters with big performance impact that the state-of-art tool [34] fails to detect, and covering most of that identify by the prior work. Also, SafeTune is more efficient and lightweight.

5.3 RQ3: Effectiveness Study in Helping Performance Tuning

SafeTune can help tuning tools (e.g., OtterTune [43], BestConfig [54]) avoid potential side-effects on other user intentions. To prove the effectiveness of SafeTune, we conduct a case study on one of the auto-tuners and manually validate the result. Among these publicly available auto-tuners, OtterTune [43] is the most popular one with 1.1k `Github` stars. OtterTune supports MySQL

```
-- Startup PostgreSQL with the default configuration (fsync=ON)
Bash# ./tpcc_run.lua &          # TPC-C, which is write intensive
Bash# kill -9 [postgresql-pid]
-- Then, restart PostgreSQL, clear cache to force reading from disk.
postgres=# SELECT * FROM test_table LIMIT 10000;
    ..... (10000 rows successfully returned)
                    (Disk: Intel P4510 SSD, with disk-failure guard)

-- run OtterTune and startup PostgreSQL with the configuration
suggested by OtterTune (fsync=OFF)

Bash# ./tpcc_run.lua &          # TPC-C, which is write intensive
Bash# kill -9 [postgresql -pid]
-- Then, restart PostgreSQL, clear cache to force reading from disk.
postgres=# SELECT * FROM test_table LIMIT 10000;
    WARNING: page verification failed, calculated checksum 39438 but expected 39327
    ERROR: invalid page in block 1 of relation base/13425/6892 (Data corruption)
```

(a) Lower reliability

```
-- Start MySQL with default configuration (innodb_buffer_pool_size=128M)
Bash# top -p [mysqld-pid]
PID   USER    PR   NI      VIRT      RES     SHR      S    %CPU  %MEM   COMMAND
4329  mysql   20    0    1.165g   0.041g   0.015g   S     0.0   0.0    mysqld

-- run OtterTune and startup MySQL with the configuration suggested
by OtterTune (innodb_buffer_pool_size=16.4G)

Bash# top -p [mysqld-pid]
PID   USER    PR   NI      VIRT      RES     SHR      S    %CPU  %MEM   COMMAND
4919  mysql   20    0   18.301g   1.340g   15152   S     0.0   1.1    mysqld
        More Cost (30 $USD/month for single node, ecs.hfc7.xlarge)
```

(b) More cost

```
Show the data files MySQL is monitoring (performance_schema=ON):
mysql> SELECT * FROM performance_schema.file_instances LIMIT 3;
    +-----------------------------------+----------------------+------------+
    | FILE_NAME                         | EVENT_NAME           | OPEN_COUNT |
    +-----------------------------------+----------------------+------------+
    | /var/lib/mysql/ibdata1            | innodb/innodb_data_file |          3 |
    | /var/lib/mysql/ib_logfile0        | innodb/innodb_log_file  |          2 |
    | /var/lib/mysql/mysql/engine_cost.ibd | innodb/innodb_data_file |          3 |
    +-----------------------------------+----------------------+------------+

-- Run OtterTune and startup MySQL with the configuration suggested
by OtterTune (performance_schema=OFF)

-- Then, user may want to monitor the status of data files via:
mysql> SELECT * FROM performance_schema.file_instances LIMIT 3;
    Empty set (0.00 sec)     # Reduced Functionality
                        (performance schema monitoring not work)
```

(c) Reduced functionality

```
-- Startup MySQL with the default configuration (innodb_flush_method=fsync)
Bash# ./tpcc_run.lua                    # User A: TPC-C workload
    Transactions:     250.26 per second
    Latency (ms) avg:  31.90
mysql> source tpch/query-14.sql;        # User B: TPC-H workload
    Query OK, 1 row in set (21.81 sec)

-- run OtterTune and startup MySQL with the configuration suggested
by OtterTune (innodb_flush_method=O_DIRECT)

Bash# ./tpcc_run.lua
    Transactions:    340.94 per second
    Latency (ms) avg: 23.40  1.36x faster for TPC-C workload (user 1)
mysql> source tpch/query-14.sql;
    Query OK, 1 row in set (1 min 29.21 sec)
                4.09x performance drop for TPC-H workload (user 2)
```

(d) Lower performance (other user)

Figure 4: Side-effects on other intentions caused by OtterTune without the aid of SafeTune.

and PostgreSQL, so we run OtterTune in these two software under default tool setting and apply SafeTune to check if SafeTune can warn potential side-effects. Further, to prove those side-effects on other user intentions do have severe consequences, we manually validate if the corresponding intentions are violated. In the case study, we assume 4 users who want to leverage OtterTune to improve performance (with their non-performance intention shown in the end):

- **User A**: military communication service provider who is obligated to preserve data reliably. – *High reliability*
- **User B**: free service provider who uses free cloud instances with limited resources. – *Low cost (resource)*
- **User C**: system administrator who is responsible to monitor unexpected behavior of the database. – *Functionality*
- **User D**: social network application provider who faces many different user requests. – *Good performance (all users)*

Result and Analysis. Overall, SafeTune warns 8 side-effects (covering 4 types, excluding "lower security" and "limited side-effect", OtterTune does not touch any security related parameters) on other user intentions caused by OtterTune. We discuss how the intentions of **User A-D** are violated by the 4 out of 8 side-effects in detail, and we put the other 4 cases in the public repository. For **User A**, `fsync` is the parameter to be affected. It is turned off by OtterTune during tuning, whose document [12] is written as: *"While turning off fsync is often a performance benefit, this can result in* ***unrecoverable data corruption*** *in the event of a power failure or system crash. Thus it is only advisable to turn off fsync if you can easily recreate your entire database from external data."* Though the workload performance improved by ~70% after tuning by OtterTune, the database get unreliable. As shown in the Fig. 4(a), we simulate an occasionally power loss when PostgreSQL is serving request normally by issuing `kill -9` to the `postgres` server process and clear the system cache which would not survive during a power loss. After restarting the process, we observe the **User A**'s data is corrupted (yellow/red text in Fig. 4(a)) This is because by turning off this parameter, PostgreSQL will only persist data once the buffer is full. We also observed that when `fsync` is turned on, the simulated power loss never causes the data corruption. By applying SafeTune, this parameter is warned clearly to have reliability impact.

Running OtterTune in MySQL, the parameter `innodb_buffer_pool_size` used by **User B** is increased from 128MB to 16.4GB as shown in Fig. 4(b). This is because using the large `innodb` buffer, more data can be cached, improving the performance. While **User B** may use MySQL in a free cloud virtual machine. Using big amount of memory leads to extra budget for ~30$ per month (depending on the cloud platform provider). While such consequence is warned by using SafeTune in prior.

For **User C**, OtterTune suggests to turn of parameter `performance_schema` because by doing so, the performance improves by ~25%. This parameter enables MySQL monitoring on various entities, including events, opened files, status information and etc. **User C** monitors unexpected behavior (e.g., too many contentions on data files) by this functionality. But after turning it off, as shown in Fig. 4(c), any monitoring action (i.e., monitor which files are being opened by how many entities) does not work. Hurting **User C**'s initial intention.

User D is affected by `innodb_flush_method`. As shown in Fig. 4(d), after running OtterTune, this parameter is tuned from `fsync` to `O_DIRECT`. The former value allows each data write first touch the kernel's cache. Since MySQL implements buffering (in user mode) itself (especially for write [10]), the kernel level caching may conflict with the MySQL buffering. And the latter value makes MySQL bypass the kernel cache. So if **User D** uses the configuration setting suggested by OtterTune, the write workload (green text in Fig. 4(d)) can be improved by ~36%. But **User D** is facing many kinds of users (i.e., workload). As the red text shown in Fig. 4(d), the read performance degrades dramatically by this configuration setting. The reason lies in the kernel cache can keep more hot data in memory, bringing speed up for read.

In conclusion, auto-tuners may cause critical side-effects on other intentions, SafeTune is complementary with them that helps to prevent the bad consequences.

6 RELATED WORK

Configuration Tuning. Some works target improving software performance by tuning configurations. They can be classified into model-based tuning [16], measurement-based tuning [24], search-based tuning [19, 37, 38, 45, 47, 54] and learning-based tuning [18, 32, 39]. These tuning tools tune the parameters by some heuristics and measure software performance to build a model that can find the fastest configuration. They only consider the performance impact of configurations, so they may cause side-effects such as reducing reliability. While SafeTune can help these tools by both accelerating them by reducing the search space and warning these side-effects to prevent severe consequences.

Pre-selecting Performance-related Parameters. Some works target per-selecting important parameters to accelerate the configuration tuning process. They use performance experiments to dynamically choose parameters that have a significant influence on performance using statistic [34] or machine learning [23] techniques. These works only focus on the performance of software and pay no attention to non-performance intentions of users like reliability, security, and functionality. While SafeTune covers their targets and aware of multi-intentions additionally. It fully leverages the document of parameters to predict the performance-related parameters and potential non-performance side-effects.

Understanding Relationship between Performance and Configuration. Some works target understanding the relationship between performance and configuration parameters. A group of works understands the relationship from code. They use static or dynamic code analysis [31, 35, 49]. While they ignore the non-performance impact of the configuration parameters, and we understand this relationship in the end-user's view – SafeTune considers both performance and non-performance user intentions. The other group of works mines useful information related to configuration to help both developers and users improve software performance. Some works mine configuration documents to detect performance bugs [29] and to identify main intentions [53], Some works mine performance constraints that are related to configuration to prevent users from performance misconfigurations [48]. Compared with all these works, SafeTune has a different focus. It utilizes configuration documents to obtain multi-intentions of configuration

parameters to pre-select performance-related parameters and warn potential side-effects.

7 DISCUSSION

Ability of Generalization. The six types of side-effects on non-performance intentions are concluded from the studied software. To make SafeTune as generalized as possible, we selected 13 widely-used open-source software systems from 4 representative categories as targets. And our evaluation in §5.1 and §5.2 shows SafeTune can achieve good results on un-studied software. But we still cannot claim that our approach can be generalized to all software domains. SafeTune leverages configuration documents to predict tuning guidance. So the tuning guidance that SafeTune provides relies on the quality of the configuration documents of the target software systems. Since the inaccuracy of SafeTune is related to the low quality, We suggest that the documents can 1) explain the context of parameters' functionality, 2) tell user what will result in by changing parameters' value, 3) split the description of parameter and the additional information (e.g., recommendations, constraints with other parameters) into different paragraphs. Our future work will extend SafeTune by using more information (i.e., source code) as additional input and techniques (e.g., static analysis) to further understand the side-effects on non-performance intentions.

Effectiveness on Reducing the Search Space for Tuning. SafeTune may produces many performance related parameters (e.g., occupying 38.9% of all parameters in §5.2) for performance tuning, directly using them may still make the search space big during tuning. However, parameters suggested by SafeTune are labeled with side-effects so that many of the parameters may not be actually tuned in tuning (given some of other user intentions as input). In fact, there are only small proportion (8.4% of all parameters in §5.2) of performance related parameters that have limited or no side-effects. Also, we argue SafeTune identifies performance related parameters in general and independent of workload. So users can further choose parameters according their workload. One of our future works lies in automatically identifying workloads that a given performance related parameter affect.

Triggering Conditions of the Side-effects. Tuning parameters with side-effects may not necessarily violate user intentions. If **User A** of §5.3 turns off `fsync` but he/she has battery-backed RAM in the event of power failure, the intention of high reliability would not be violated. Such conditions (e.g., with/without battery-backed RAM) may come from system environment, production workload, malicious attacks, etc. SafeTune is not able to extract all triggering conditions, and we claim SafeTune warns potential side-effects on other user intentions.

8 CONCLUSION

Users change configuration parameters to satisfy their intentions like better performance, reliability, etc. Existing works automatically pre-select & tune the parameters to improve performance but only for specific workloads and are unaware of other user intentions. We argue that configuration document has rich information and can be leveraged to provide guidance on pre-select important parameters in general while keeping other non-performance intentions. We conclude six types of non-performance side-effect of the performance-related parameters from an empirical study on 13 software systems. Based on the finding, we design and implement SafeTune to predict the tuning guidance. The experiments show that SafeTune can identify 28 performance related parameters that with big performance impacts but missed by state-of-the-art tools. And SafeTune can help the auto-tuner prevent 8 potential side-effects on other user intentions that can cause severe consequences.

ACKNOWLEDGMENTS

This paper is supported by National Key R&D Program of China (Project No.2018YFB0204301); National Natural Science Foundation of China (Project No.61872373); Guangdong Major Project of Basic and Applied Basic Research (Project No.2019B030302002); The Major Key Project of PCL.

REFERENCES

[1] 1988. GCC, the GNU Compiler Collection. Retrieved 2021 from http://gcc.gnu.org/
[2] 2014. StackOverFlow #27176623. https://stackoverflow.com/questions/27176623/
[3] 2016. MySQL 8.0 Reference Manual :: 15.14 InnoDB Startup Options and System Variables. Retrieved 2021 from https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_flush_log_at_trx_commit
[4] 2017. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)* (2017), 1–541.
[5] 2020. Category:Computing - Wikipedia. Retrieved 2021 from https://en.wikipedia.org/wiki/Category:Computing
[6] 2021. sklearn.metrics.average_precision_score–scikit-learn 0.24.2 documentation. Retrieved 2021 from https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html
[7] Accessed 2021. AWS Cloud Products. https://aws.amazon.com/products
[8] Accessed 2021. The cassandra-stress tool. https://cassandra.apache.org/doc/4.0/cassandra/tools/cassandra_stress.html
[9] Accessed 2021. Google Cloud products. https://cloud.google.com/products
[10] Accessed 2021. Optimizing InnoDB Disk I/O: store system tablespace files on Fusion-io devices. https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-diskio.html
[11] Accessed 2021. Oracle Cloud Infrastructure Products by Category. https://www.oracle.com/cloud/products.html
[12] Accessed 2021. PostgreSQL documentation. https://www.postgresql.org/docs/13/index.html
[13] Accessed 2021. Principal component analysis. https://en.wikipedia.org/wiki/Principal_component_analysis
[14] Accessed 2021. Transaction Processing Performance Council Benchmark C (TPC-C). http://www.tpc.org/tpcc/
[15] Accessed 2021. Transaction Processing Performance Council Benchmark H (TPC-H). http://www.tpc.org/tpch/
[16] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. 2004. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering* 30, 5 (2004), 295–310.
[17] Liang Bao, Xin Liu, Fangzheng Wang, and Baoyin Fang. 2019. ACTGAN: automatic configuration tuning for software systems with generative adversarial networks. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 19)*. IEEE, 465–476.
[18] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. 2018. Autoconfig: Automatic configuration tuning for distributed message systems. In *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 18)*. IEEE, 29–40.
[19] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research* 13, 1 (2012), 281–305.
[20] L Breiman. 2001. Random Forests. *Machine Learning* 45 (2001), 5–32.
[21] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.
[22] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. 2016. *A dictionary of computer science*. Oxford University Press.
[23] Zhen Cao, Geoff Kuenning, and Erez Zadok. 2020. Carver: Finding important parameters for storage system tuning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*. 43–57.

[24] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 16)*. 49–60.

[25] Chuancong Gao, Jianyong Wang, Yukai He, and Lizhu Zhou. 2008. Efficient mining of frequent sequence generators. In *Proceedings of the 17th international conference on World Wide Web (WWW 08)*. ACM, 1051–1052.

[26] Yoni Gavish, Jerome O'Connell, Charles J Marsh, Cristina Tarantino, Palma Blonda, Valeria Tomaselli, and William E Kunin. 2018. Comparing the performance of flat and hierarchical Habitat/Land-Cover classification models in a NATURA 2000 site. *ISPRS Journal of Photogrammetry and Remote Sensing* 136 (2018), 1–12.

[27] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. 2005. Borderline-SMOTE: A New over-Sampling Method in Imbalanced Data Sets Learning. In *Proceedings of the 2005 International Conference on Advances in Intelligent Computing (ICIC 05)*. 878–887.

[28] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: Learning from Bug Reports to Understand and Generate Performance Test Frames. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 18)*. 17–28.

[29] Haochen He, Zhouyang Jia, Shanshan Li, Erci Xu, Tingting Yu, Yue Yu, Ji Wang, and Xiangke Liao. 2020. CP-Detector: Using Configuration-related Performance Properties to Expose Performance Bugs. In *Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 20)*. IEEE, 623–634.

[30] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. *spaCy: Industrial-strength Natural Language Processing in Python.* https://doi.org/10.5281/zenodo.1212303

[31] Yigong Hu, Gongqi Huang, and Peng Huang. 2020. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 719–734.

[32] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 17)*. IEEE, 497–508.

[33] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 12)*. 77–88.

[34] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[35] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys 20)*. 1–16.

[36] Andy Liaw and Matthew Wiener. 2002. Classification and Regression by randomForest. *R News* 2, 3 (2002), 18–22.

[37] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering and Symposium on the Foundations of Software Engineering (ESEC/FSE 17)*. 257–267.

[38] Rafael Olaechea, Derek Rayside, Jianmei Guo, and Krzysztof Czarnecki. 2014. Comparison of exact and approximate multi-objective optimization for software product lines. In *Proceedings of the 18th International Software Product Line Conference (SPLC 14)*. 92–101.

[39] Cheng Peng, Canqing Zhang, Cheng Peng, and Junfeng Man. 2017. A reinforcement learning approach to map reduce auto-configuration under networked environment. *International Journal of Security and Networks* 12, 3 (2017), 135–140.

[40] PostgreSQL Global Development Group. 2008. PostgreSQL. http://www.postgresql.org.

[41] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[42] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the 26th symposium on mass storage systems and technologies (MSST 10)*. IEEE, 1–10.

[43] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*. 1009–1024. https://github.com/cmu-db/ottertune

[44] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 18)*. 154–168.

[45] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering and Symposium on the Foundations of Software Engineering (ESEC/FSE 13)*. 455–465.

[46] Duane Wessels, Henrik Nordström, Amos Jeffries, Alex Rousskov, Francesco Chemolli, Robert Collins, and Guido Serassio. 1996. Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[47] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO 15)*. 1375–1382.

[48] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 265–280.

[49] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 619–634.

[50] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.

[51] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data (ICMD 19)*. 415–432.

[52] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 21)*. 175–176.

[53] Chenglong Zhou, Haoran Liu, Yuanliang Zhang, Zhipeng Xue, Qing Liao, Jinjing Zhao, and Ji Wang. 2021. Deep Understanding of Runtime Configuration Intention. *International Journal of Software Engineering and Knowledge Engineering* 30, 05 (2021), 1–28.

[54] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC 17)*. 338–350.