Who is in Charge here? Understanding How Runtime Configuration Affects Software along with Variables&Constants

Chaopeng Luo*‡ , Yuanliang Zhang*‡ , Haochen He†§,Zhouyang Jia*,
Teng Wang¶, Shulin Zhou*, Si Zheng*, Shanshan Li*§
*National University of Defense Technology, Changsha, China
†Key Laboratory of Satellite Information Intelligent Processing and Application Research, Beijing, China
¶National Innovation Institute of Defense Technology, Beijing, China
Email:{luochaopeng18, zhangyuanliang13, hehaochen13, jiazhouyang, wangteng13, zhoushulin}@nudt.edu.cn
si.zheng1009@gmail.com,shanshanli@nudt.edu.cn

Abstract—**Runtime misconfiguration can lead to software performance degradation and even cause failure. It is usually caused by invalid parameter values set by users. Developers typically perform sanity checks during the configuration parsing stage to prevent invalid parameter values. However, we discovered that even valid values that pass these checks can also lead to unexpected severe consequences. Our study reveals the underlying reason: the value of runtime configuration parameters may interact with other constants and variables when propagated and used, altering its original effect on software behavior. Consequently, parameter values may no longer be valid when encountering complex runtime environments and workloads. Therefore, it is extremely challenging for users to properly configure the software before it starts running.**

This paper presents the first comprehensive and in-depth study (to the best of our knowledge) on how configuration affects software at runtime through the interaction with constants, and variables (PCV Interaction). Parameter values represent user intentions, constants embody developer knowledge, and variables are typically defined by the runtime environment and workload. This interaction essentially illustrates how different roles jointly determine software behavior. In this regard, we studied 705 configuration parameters from 10 large-scale software systems. We reveal that a large portion of configuration parameters interact with constants/variables after parsing. We analyzed the interaction patterns and their effects on software runtime behavior. Furthermore, we highlighted the risks of PCV interaction and identified potential issues behind specific interaction patterns. Our findings expose the "double edge" of PCV interaction, providing new insights and motivating the development of new automated techniques to help users configure software appropriately and assist developers in designing better configurations.

Index Terms—**Software Configuration, Empirical Study, Software Performance, Software Reliability**

I. Introduction

Runtime configuration is used at deployment time to port a software system to accommodate different environments and workloads without re-compiling the software [1], [2]. Despite the convenience, configuration issues have also become one

Fig. 1: An example of a valid parameter value causes software crash during interaction.

of the major problems of large-scale software systems [3]–[5]. For instance, in October 2021, An internal configuration error caused Facebook, Instagram, and WhatsApp to go offline across the globe, affecting nearly 200 million daily users [6].

Many works [3], [4], [7], [8] find that misconfiguration can be the culprit, and developers constantly add sanity checkers to check basic syntax constraints (e.g., not-NULL, value type) [1]. However, these checks usually only work during configuration parameter parsing, which is the very early stage of the configuration usage lifecycle. **After parsing, the configuration parameter will interact with the program in various ways to achieve different goals (e.g., allocating resources and adapting the environment).**

While the complex interaction is a black box to the users, making users hard to accurately predict the effect of configuration, thereby introducing configuration-related issues. In fact, many cases [9]–[14] have shown that syntactically or even semantically valid parameter values can lead to serious consequences.

---

‡ Chaopeng Luo and Yuanliang Zhang are co-first authors.
§ Haochen He and Shanshan Li are the corresponding authors.

Figure 1 shows an example [10] from HBASE that a valid parameter value may cause a software crash due to interaction with program workload. The parameter *jute.maxbuffer* restricts the size of data in a single *znode*. In this case, a user sets it with the default value of 1MB (which is the best out-of-box value). However, when a *regionServer* manages hundreds or even thousands of regions, the data volume from backed-up Write-Ahead Logs (WALs) during recovery can reach hundreds or even thousands. The 1MB buffer cannot handle such a huge workload, causing cluster crashes. After inspecting the source code, this parameter value interacts with the variable "*len*" (representing the workload). If the variable's value exceeds the configured threshold, a runtime exception will be thrown. The reporter suggested resolving the problem by increasing the default value of this parameter. However, such an approach cannot solve the problem completely as the software will face an even larger workload.

Considerable effort has been made to address configuration issues [2], [4], [15]–[18]. However, Existing work mainly focuses on detecting invalid configuration values or flawed configuration parsing code. They lack a comprehensive and deep understanding of how configuration affects software behavior, especially in terms of the interaction between configuration parameters and other program constants or variables. Configuration parameter values represent user intention, while constants incorporate developer knowledge and variable values are usually determined by runtime environment/workload. Understanding such interactions can help developers learn good practices and realize potential risks that configuration may cause. Additionally, it can help users set parameter values with greater precision (beyond simply "valid") by elucidating the impact of these parameters on performance and reliability.

In this paper, we conduct the first comprehensive analysis (to the best of our knowledge) of the interaction of parameter & constant & variable (**PCV Interaction**) at the source code level to understand how configuration affects software at runtime. We organize the study by answering four research questions:

- **RQ1:** Are PCV interactions common in software?
- **RQ2:** What are the types and patterns of PCV interactions?
- **RQ3:** How do PCV interactions take effects on software?
- **RQ4:** What are the potential problems behind PCV interactions?

To achieve this, we study 705 configuration parameters from 10 extensively used, large-scale software systems, tracing parameter parsing, propagation, and usage in the source code and manually analyzing the results. We find that a large portion of configuration parameters (66.4%) will interact with other constants/variables during propagation and usage, and the interactions can have a great impact on software performance and reliability. We conclude seven types of interaction patterns (§III). These interactions can affect software behaviors in four main aspects: misconfiguration prevention, environment adaption, workload adaption, and fault tolerance (§IV). We

further analyze and identify potential problems behind four main interaction patterns, providing future research directions with actionable suggestions for each identified issue (§V).

Overall, this paper makes the following contributions:

- **New perspective of how configuration affects software at runtime.** We study how configuration affects software at runtime from the view of PCV interaction, which introduces three different roles: user intention, developer knowledge, and runtime environment/workloads. We comprehensively analyze the patterns, effects, and potential problems of PCV interaction.
- **Findings and insights.** We present novel findings and insights derived from our study. Our findings and insights reveal the "double-edge" of PCV interactions, which can help developers and users to better design and use configuration. Also, we point out several research directions for future configuration automated tools.
- **Dataset.** We release our dataset in this paper for future research in this area. Our dataset can be used for extensive study and tool assessment. The dataset is available at: https://github.com/PCVAnonymous/PCVStudy

II. STUDY METHODOLOGY

To study the interaction between constants and variables in configurations and programs. We aim at source code level study, which is the most fine-grained way to reveal the interaction relationship. We start with configuration parameters and then find their related code snippet and background knowledge (commits and issues).

A. Selecting Software and Configuration Parameters

*a) **Target software**:* To ensure the representation of our study results, we choose 10 software systems from different domains, including database, file system, data processing, and web server, as shown in Table I. We selected these software systems because they 1) are mature, widely-used software systems in different fields, 2) have a large number of configuration parameters with detailed user manuals and documentation, and 3) are actively developed with well-organized GitHub repositories and bug databases.

*b) **Configuration parameters**:* Manually studying and analyzing all parameters is impractical. We first collect parameters in the default configuration file as they are visible to all users (some parameters are hidden in source code and only used by experts or developers) and get 3523 parameters. Subsequently, we randomly sample 20% of them and finally study 705 parameters across all software systems. The distribution is shown in Table I.

B. Taint Analysis of Configuration Parameters

In order to find the PCV code snippets, we need to trace the propagation and utilization of configuration parameters within the source code. We leverage two open-source taint analysis tools for: Conftainter [19] for C and C++ software and CFlow [20] for Java software.

TABLE I: Software selected in our study

SOFTWARE	DESC.	LANG.	#P	#PS
Httpd	Web server	C	557	111
PostgreSQL	Database	C	251	50
Nginx	Proxy server	C	480	96
MySQL	Database	C++	390	78
HBase	Database	Java	174	35
Hive	Database	Java	484	97
HDFS	File system	Java	463	93
Yarn	Resource manager	Java	450	90
MapReduce	Data processing	Java	168	34
ZooKeeper	Config manager	Java	154	21
Total	-	-	3523	705

Typically, the lifecycles of parameters in source code include three main stages, i.e., parsing, propagation, and usage. After setting parameters in command lines or parameter files, the values will be parsed and stored in config variables. If users do not set the value, the default value will be used in the program. The parameter value will be propagated along data-flow using the Use-Define Chain, arithmetic operations (intra-procedural), or referenced by a function (inter-procedural). Previous work predominantly concentrated on the propagation of configurations within the software itself (including Conf-tainter and Cflow). After the parameter propagates through a series of stages, it will ultimately be utilized in a conditional branch or be referenced by an external function (e.g., syscall and STL function), and then we will stop tracing.

C. Analyzing Interaction Cases. (RQ1)

The interaction of variables in a program is natural. **The criterion of PCV interaction is whether the configuration value will be affected by other program constants or variables before it is finally used and takes effect.** In this paper, we determine whether a PCV interaction exists by checking whether the configuration parameter engages in arithmetic or logical operations with constants or variables during its propagation and usage.

We collect the statement where this interaction occurs. If a configuration parameter undergoes multiple interactions during propagation, we will collect all statements where such interactions occur. For a configuration parameter that doesn't undergo any interactions, we consider it as direct usage. Finally, we collected 851 cases (including both interactions and direct usages) for 705 parameters. We manually study each case. We use `git-blame` to find out the commit that introduces the code and study corresponding issues on JIRA [21] or GitHub [22]. We analyze the issue description, and developers' discussion along with source code to understand the background and knowledge behind the interaction.

We find only about one-third (237 / 705 = 33.6%) of parameters can independently affect software behavior without conducting PCV interactions.

> **Finding 1: Only one-third (33.6%, 237 / 705) of parameters can independently affect software behavior by propagating and using its value faithfully.** The rest of the parameters will interact with constants or variables.

III. PCV INTERACTION PATTERN (RQ2)

To better understand the interaction relationships of PCV, we analyze the code snippets of PCV interactions. There are four main interaction types: 1) Parameters work independently (P), 2) Parameters interact with Constants ($P\&C$), 3) Parameters interact with Variables ($P\&V$) and 4) Parameters interact with Constants and Variables ($P\&C\&V$). We further subdivide them and get seven interaction patterns as shown in Table II. We collect 851 data points from the sample of 705 parameters, the statistical results are shown in Table III.

TABLE II: The patterns of PCV Interaction

TYPE	PATTERN	DOMINANT RELATION	ABBR.
P	$\circ = p$	P is self-dominant	P
P&C	$P = f(p,c)?p:\phi$	P is constrained by C	$C \to P$
	$\circ = g(p,c)$	P, C take equivalent place	$P - C$
P&V	$P = f(p,v)?p:\phi$	P is constrained by V	$V \to P$
	$V = f(p,v)?v:\phi$	P restrains V	$P \to V$
	$\circ = g(p,v)$	P, V take equivalent place	$P - V$
P&C&V	$\circ = g(p,c,v)$	P, V, C take equivalent place	P–V–C
	$P\&C \wedge P\&V$	Mixed	/

[1] f() is a comparison function and g() is an arithmetic function.
[2] $\phi \in \{p, c, v, g(p,c), g(p,v), \textit{ThrowException}\}$

A. Parameter works independently (P)

If a configuration parameter, during the process from being parsed to the usage phase, does not interact with other variables or constants, we consider the configuration parameter is independently affecting software behavior. Note that if multiple parameters interact with each other (also known as parameter dependency [23]), the interacted parameters belong to this type because they all represent user-set configuration. Figures 2a illustrate an example of configuration parameters independently affecting software behavior. Before being called by external functions (*defaultPool()*), *opThreads* do not interact with any variables or constants.

B. Parameter interacts with Constant (P&C)

a) ***Parameter is constrained by Constant ($C \to P$):*** In this scenario, constants serve to constrain the range of parameters. If the user-defined parameter values fall outside this range, the program will handle this situation.

b) ***Parameter and Constant take equal place ($P - C$):*** In this scenario, parameters and constants collaboratively determine (denoted as "$-$") the value of another variable or the execution path of the branch through numerical computations, devoid of any constraints between them. Taking Figure 2c as an example, In other words, the parameter (*srv_page_size*) and the constant (*IO_SIZE*) jointly determine the size of I/O space (*min_io_size*) with equivalent status.

```
                                    /* HBase/RpcServer.java */
int opThreads = conf.getInt(…);
ThreadPoolExecutor pool =
  ProcedureMember.defaultPool(…, opThreads, …);
```

```
                                  /* Httpd/mod_md_config.c */
if (retry_failover < 0) {
  return "invalid argument, must be a number > 0";
}
```

```
                                /* MySQL-server/ddl0ctx.cc */
constexpr size_t IO_BLOCK_SIZE = 4 * 1024;
min_io_size = (srv_page_size / 2) + IO_BLOCK_SIZE;
```

(a) P   (b) C→P   (c) P-C

```
                        /* Hive/PartitionManagementTask.java */
final ExecutorService executorService =
  Executors.newFixedThreadPool(
  Math.min(candidates.size(), threadPoolSize),…);
```

```
                              /* HDFS/CacheManager.java */
if (numListed++ >= maxListCachePoolsResponses) {
  …
  return new BatchedListEntries<CachePoolEntry>(…);
}
```

(d) V→P   (e) P→V

```
                        /* HDFS/FSImageFormatProtobuf.java */
if (loadInParallel && (stageSubSections.size()> 0))
  inodeLoader.loadINodeSectionInParallel(…);
else
  inodeLoader.loadNodeSection(…);
```

```
                            /* Nginx/ngx_http_core_module.c */
if(r->headers_in.safari&&(clcf->keepalive_disable
    & NGX_HTTP_KEEPALIVE_DISABLE_SAFARI)) {
  …
}
```
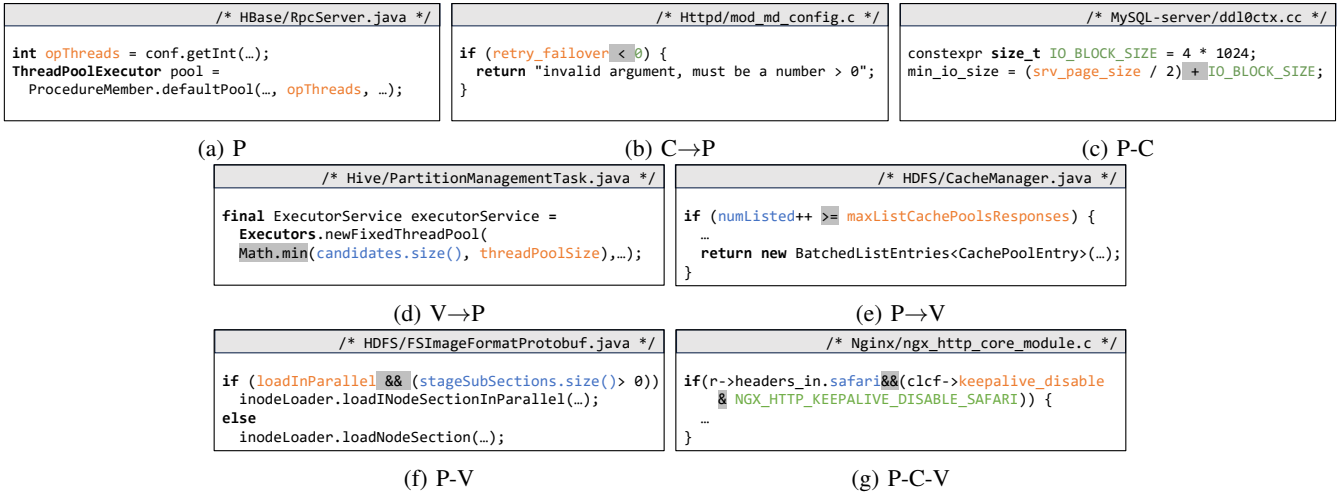
(f) P-V   (g) P-C-V

Fig. 2: Examples of PCV interaction patterns. Parameters, variables, and constants are marked in orange, green, and blue respectively, and the interactions code are highlighted in gray.

TABLE III: Data distribution of seven interaction patterns

SOFTWARE	P	C→P	P-C	V→P	P→V	P-V	P-C-V	TOTAL
HBase	10	12	0	4	15	4	0	45
HDFS	28	30	1	9	32	18	0	118
Hive	37	22	0	15	25	15	1	115
Httpd	37	32	4	0	25	26	1	125
MapReduce	12	9	0	5	16	5	0	47
MySQL	16	18	9	18	27	9	3	100
Nginx	37	12	10	6	26	11	6	108
Postgres	15	17	3	2	16	8	0	61
Yarn	38	23	0	5	30	8	0	104
Zookeeper	5	9	0	2	10	1	1	28
Total	235	184	27	66	222	105	12	851

C. Parameters interacts with Variables (P&V)

a) Parameter is constrained by Variable(V→P): In this scenario, the parameter value will be compared with the variable value. Using Figure 2d as an illustration. If the parameter *threadPoolSize* exceeds the value of *candidates.size()*, the variable will overwrite the parameter value and determine the size of the thread pool.

b) Parameter restrains Variable (P→V): Parameters serve as thresholds at conditional branches, constraining the values of variables. Figure 2e illustrates an example from HDFS. The parameter *maxListCachePoolsResponses* controls the number of cache pools that the `NameNode` will send over the wire in response to a `RPC`, i.e., *numListed*.

c) Parameter and Variable take equal place (P−V): Variables and parameters can also engage in interactions similar to *P−C*, where they take equal place. In Figure 2f, the program runs the parallel function in L4 only if the parameter and variable values are true.

D. Parameter interacts with Constant & Variable (P&C&V)

a) Parameter, Constant and Variable take equal place (P-C-V): Parameters, constants, and variables may appear in the same statement for arithmetic or logical operations. For instance, in Figure 2g, the parameter *keepalive_disable*, variables *safari*, and constant

NGX_HTTP_KEEPALIVE_DISABLE_SAFARI collectively determine the execution path of the branch.

b) The combination of P&C and P&V: During the propagation, parameters may interact initially with either a constant or a variable, and subsequently with the other. In these scenarios, we will split the interaction into two cases, i.e., P&C and P&V. We will then separately investigate the interaction effects and problems of them. Here is an example in HDFS:

```
1  /* HDFS/RpcHandler.java */
2  if ((nodeLoad > maxLoad) && (maxLoad > 0)) {
3    ...
4    return true;
5  }
```

The configuration parameter `maxLoad` interacts with both the constant `0` and the variable `nodeLoad` in the same assignment. We will split this assignment into two cases.

IV. THE EFFECT OF PCV INTERACTION ON SOFTWARE AT RUNTIME (RQ3)

As PCV interactions are inevitable in software running, it is important to understand how they take effects on software at runtime. To answer this question, we first investigate the overall impact of PCV interactions, mainly from the perspective of affecting software performance and reliability (§IV-A). To achieve performance and reliability purposes, we analyze how developers manipulate PCV interactions to control software behavior at runtime (§IV-B).

A. Overall Impact

We analyze the overall impact of PCV interaction on software from the perspective of performance and reliability, which are extremely important when providing long-term services. In this section, we study the positive sites. The potential problem will be discussed in Section V. We find that 38.5% (237/616) of the PCV interaction will affect software performance and 41.7% (257/616) will affect reliability. While the others are usually used for functionality. For different
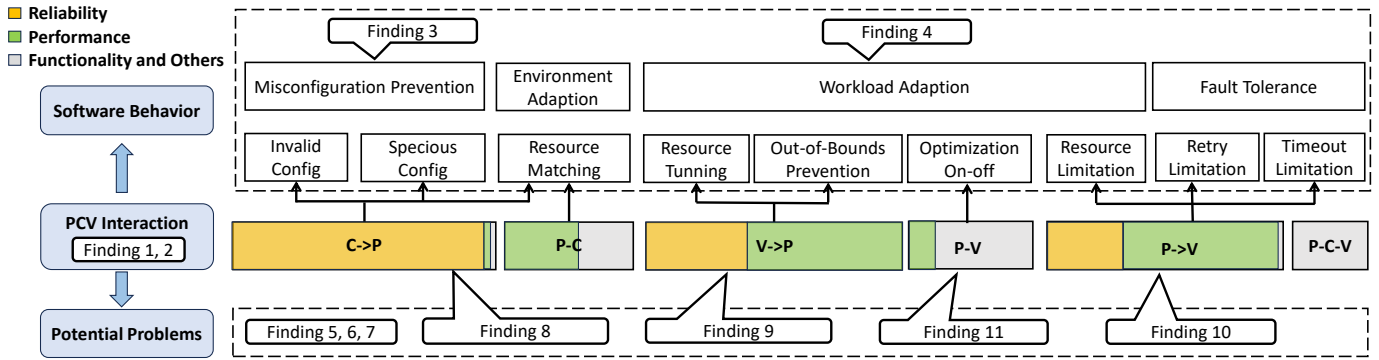
Fig. 3: PCV interaction: Patterns, Effects, and Problems (the colors represent the proportion).

interaction patterns, the impact degree is divided by three colors according to the percentage (Figure 3).

First, PCV interaction can be used to improve software performance. The following code snippet shows an example:

```
1 /* MySQL/ddl0ctx.cc */
2 size_t Context::merge_io_buffer_size(...) noexcept {
3   ...
4   return std::max(std::max((ulong)srv_page_size,
5       (ulong)IO_BLOCK_SIZE), (ulong)io_size);
6 }
```

The variable *io_size* records the size of the IO buffer during the file loading phase in MySQL, and *IO_BLOCK_SIZE* (4096) is a constant value. If the user-configured value for *srv_page_size* is too small, it may result in misalignment with the IO block and the inability to promptly process loaded files. To ensure software performance, the final size of the IO buffer will be determined by selecting the maximum value among the three.

PCV interaction can also improve software reliability. We show an example in HDFS:

```
1 /* HDFS/DiskBalancer.java */
2 if (item.getErrorCount() <= getMaxError(item))
3   item.setErrMsg("Error count exceeded.");
```

The configuration parameter *maxDiskErrors* controls the maximum errors allowed by the HDFS balancer when moving data blocks between two disks. At the branching point, the parameter interacts with *getErrorCount()* to control the error count and promptly enable the replication disk, ensuring the software reliability.

> **Finding 2: PCV interactions can have a great impact on software performance (38.5%) and reliability (41.7%).** Others are usually the implementation of logical functionalities.

B. Controlling Software Behaviors

In practice, the way that PCV interaction affects performance and reliability is by controlling software behavior at runtime. We conclude four main aspects which are misconfiguration prevention, environment adaption, workload adaption, and fault tolerance. As shown in Figure 3, different interaction patterns can influence software behavior in multiple detailed ways.

1) Misconfiguration Prevention: Misconfiguration consists of two parts: one involves invalid parameter values that do not meet the program's hard-coded constraints [4], and the other comprises specious parameter [24] that might lead to poor performance and reliability issues in the software. Figure 2b gives an example of using C→P to prevent misconfiguration. If *retry_failover* is smaller than 0, then the method will return, preventing the invalid value from propagating and using.

We find that the most of P&C interaction cases (81.5%, 172/211) fall into this category. In these cases, the constants incorporate developer knowledge as they are pre-defined in the coding process. Developers understand the functional logic of software and the design of configuration better than users, so they are more likely to prevent misconfiguration by leveraging proper constants as checkers.

> **Finding 3: P&C interactions are primarily (81.5%, 172/211) used for misconfiguration prevention.** A good constant used as a checker should incorporate developer knowledge.

2) Environment Adaption: Parameters adapt to the environment by interacting with pre-defined constants set by developers, primarily focusing on resource matching to the hardware environment.

Mismatching may arise between parameters and the physical hardware, such as misalignment of I/O blocks or conservative settings of I/O throughput [25]. To address such issues, developers proactively define constants to interact with parameters. In Figure 2c, developers discovered that using half the value of parameters as the buffer size may lead to misaligned I/O blocks, resulting in performance degradation [26]. During maintenance, developers augmented the original buffer size with a constant, *IO_BLOCK_SIZE*, to ensure a proper I/O block alignment.

3) Workload Adaption: The workload at runtime is constantly changing. However, parameter values will remain static if they are not updated after parsing. In this situation, developers make parameters adapt to the workload by interacting with workload-related variables. This primarily involves three modes: variables replace parameters that are unsuitable for the

workload (V→P), parameters and variables jointly determine the value that software uses at runtime (P-V), and parameters serve as thresholds to limit the value of variables (P→V).

Take Figure 2d as an example, parameter *threadPoolSize* controls the number of threads in partition management of Hive. If the parameter value exceeds the number of candidate tables, the size of threadpool will be determined by the value of *candidates.size()*. This approach can save a certain amount of resources, enhancing the overall performance of the software.

We find a large portion of P&V interactions (66.2% (260/393)) fall into this category. The reason is that variables often record workload information. By interacting with these workload-related variables, configuration values can be modified to adapt to runtime workload. Specifically, we find 76.2% (198/260) of them are to prevent excessive resource utilization. One situation is that when resources are limited, the resources beyond the limitation will restricted. Another situation is that when a task does not require too many resources, the P&V interaction will ensure that resource allocation will not be wasted. The former ensures the reliability of the software, while the latter prevents resource-induced performance issues.

> **Finding 4: 66.2% (260/393) of P&V interactions are used for workload adaption.** Among them, 79.2% (198/260) are to prevent excessive resource utilization, which have important effects on both software performance and reliability.

4) Fault Tolerance: When an internal error occurs, software typically waits for a certain period and then initiates a retry to maintain the availability of service. The timeout and retry number are usually determined by parameters, which are compared in conditional branches with variables representing the current retry count, constraining the software's retry attempts. Here, we show a code example:

```
1 /* Yarn/FederationActionRetry.java */
2 if (++retry > retryCount) {
3   LOG.info("Maxed out Federation retries. Giving up!");
4   throw e;
5 }
```

V. POTENTIAL PROBLEMS OF PCV INTERACTIONS (RQ4)

A double-edged sword cuts both ways. Despite the positive effects of PCV Interactions discussed in § IV, we also notice that there may be some hidden issues behind those interactions. To answer RQ3, we manually analyze the source code, the commits (using git-blame), and issues introducing the code. We identify the risk during the process of configuration interactions (§ V-A), as well as potential problems behind specific interaction patterns (§ V-B).

A. The "Risk" under PCV Interactions

PCV interaction has the following three main issues: first, the configuration changes its own original effect during the interaction, which may lead to the violation of the user's intention and severe consequences (e.g., software crash). At the same time, there is a lack of logs to explicitly provide

feedback to the user on key information of the interactions, which makes it difficult for users to make good adjustments to the configuration issue. Finally, we find that the software (especially Java programs) does not provide a comprehensive on-the-fly update mechanism for configuration parameters that have interactions. Thus, configuration issues in long-term services cannot be repaired at runtime, which seriously damages software reliability and availability.

1) Prone to bad consequences: PCV interactions can cause bad consequences. For example in HBASE-19660 [12], when blocking store file number (V) exceeds the value of *hbase.hstore.blockingStoreFiles* (P), all write will be stopped for more than one minute, and this can happen a few times back-to-back.

The risks mainly exist in two ways: On the one hand, users' configured values may be overwritten (even when the value is valid), thereby violating the user's intent and producing unexpected results. On the other hand, if parameters fail to meet specific constraints when interacting, the interaction condition may be broken and could lead to performance degradation and runtime error. Figure 1 in the introduction gives a real-world example. Even if the parameter value is set appropriately when the service starts, the interaction relationship may be reversed as the workload increases. In all these situations, the interactions will damage the software services at runtime.

> **Finding 5: More than half (56.5%, 349/616) of the interactions have the potential to cause bad consequences.** This including runtime error (13.7%), performance degradation (53.3%) and unexpected results (33.0%).

Research Direction: Developers and users should all be careful of PCV interactions. For developers, adequate testing for configuration interaction code should be considered. Existing fuzzing frameworks can generate configuration values as seeds to achieve higher code coverage [27]. However, as PCV interactions have multiple patterns, using a simple strategy to inject configuration values cannot cover all patterns of PCV code. Future work should investigate seed mutation strategies for configuration codes and cover as many PCV codes as possible.

2) Lack of log information: Informative log can assist users in understanding and adjusting configuration [28], [29]. Therefore, we study whether the interaction information, especially the runtime environment/workload information will be informed to the user.

We find that, unlike error logging, logs that are related to PCV interactions are seriously inadequate. Only a small portion (<25%) of interactions are conveyed to users through logs. Moreover, in those cases where logs are available, about a third (31.9%) of them are checking invalid parameter values. There is rare information provided to users regarding workload-related variables. Here, we show an example of a good log:

```
1 /* HBase/SplitTableRegionProcedure.java */
```

```
2   LOG.info("pid=" + getProcId() + " splitting " + nbFiles
        + " storefiles, region=" + getParentRegion().
        getShortNameToLog() + ", threads=" + maxThreads );
```

The log message explicitly points out *maxThreads* (P), *nbFiles* (V), and their semantic relationship. This can help users configure software better.

> **Finding 6**: **Existing logs are lack of comprehensive interaction details (or even missing) to assist users in configuration adjustments.** Only a small number of cases (15.3%) explicitly inform users about the interaction information.

Research Direction: The same as error logging, developers should provide detailed log information about PCV interactions. Future work should automatically extract the interaction using program analysis. Based on the code patterns (§ III-D) and the corresponding effects (§ IV) of the interaction code, the developers should provide detailed log information to users for configuration assistance and problem diagnosis.

3) Lack of on-the-fly update support: Restarting a software system that has been in service for a long time can result in significant overhead and unavoidable damage. Given the potential issues arising from PCV interactions (§ V-A1), we investigate whether users can mitigate and address these problems without service restart. The most effective and common approach is on-the-fly configuration update.

We find that many parameters in C/C++ software (325/335) can be updated at runtime, corroborating the results in previous work [2]. Surprisingly, the number is much lower in Java programs. Only 3.0% (7/240) of parameters support on-the-fly update. In this case, it is rather difficult for users to resolve configuration issues timely and with minimal cost.

> **Finding 7**: **The Java software does not provide sufficient support for on-the-fly update of interacted parameter.** Few of Java software parameters (7/240) support updates at runtime.

Research Direction: Developers need to provide sophisticated on-the-fly update mechanism (especially for Java programs). This is to ensure configuration issues caused by interaction at runtime can be resolved without a service restart. The difficulty here may lie in how to ensure that the updated values are propagated and used correctly and are not affected by historical status.

B. The "Crisis" Behind Specific Interactions

The main purpose for developers to introduce PCV interactions is to help the software adapt to different environments and workloads. However, facing real complex and changing environments/workloads at runtime, developers sometimes can not guarantee the stable execution of every type of interaction, and thus introduce configuration issues. In this section we discuss the "crisis" we found behind several interaction patterns, which are: Unexplained constant used in C→P, Partial

adaption to workload in V→P, Inappropriate threshold in P→V and Missing optimization opportunities in P-V.

1) Unexplained constant used in C→P: As discussed in § IV, C→P interactions are usually used for misconfiguration prevention. In this situation, developers may conditionally overwrite the parameter value with a pre-defined constant. However, we find that those constants are not carefully selected and checked. 56.7%, 34/60 of the constants used for parameter overwriting are not the default value (which is, theoretically the best out-of-box value for parameter). The code snippet shows an example:

```
1   /* MySQL/srv0srv.cc */
2   static MYSQL_SYSVAR_ULONG(..., srv_purge_batch_size,
        300 , ...)
3   const uint temp_batch_size = 20 ;
4
5   n_pages_purged = trx_purge(1,
6     srv_purge_batch_size <= temp_batch_size ?
7       srv_purge_batch_size : temp_batch_size,true);
```

In this case, the default value of *srv_purge_batch_size* is 300. However, any values greater than a "magic number" 20 will be changed to 20. Quoted from developer's comment: "*a large batch size can cause a significant delay in shutdown, so reducing the batch size to magic number 20, which we hope will be sufficient to remove all the undo records .*" [30]

We also find two cases in which parameters are mandatory overwritten by constant without conditional checking. In such cases, the parameter is a "fake" configuration. For example in *srv0srv.cc* in MySQL, developers observed that user-configured values of *lock_wait_timeout* are usually set too low. So they enforced the configuration value to one year without informing users about the change [31]. This one-year timeout could lead to prolonged blocking of transactions waiting for lock resources, requiring users to employ additional deadlock detection. Ideally, the default value is the best constant to face most scenarios. Such inconsistency could confuse users and may violate user intention.

> **Finding 8: The constants used for parameter overwritten in C→P are not carefully selected and checked.** Among the constants used for parameter overwritten, many (31/56 = 55.4%) of them differ from the parameter's default value. There are also a few "fake configurations" that the parameter will be mandatory overwritten without checking conditions.

Research Direction: Developers should be careful of using constants for parameter default value and runtime overwritten. A systematic measurement approach and test process are needed to find good constants. And making it consistent for default value and runtime parameter is overwritten. Tools should automatically find parameters that are rarely used and replace them with constants, rather than leaving a "fake" configuration.

2) Partial adaption to workload in V→P: V→P are usually used to adapt to runtime workload. During the inter-action, developers conditionally adjust parameters according

```
                                          /* MySQL#31965404 */
for (size_t i = 0; i < undo::spaces->size(); ++i) {
    ...
-   const auto n_pages = SRV_UNDO_TABLESPACE_SIZE_IN_PAGES; // Use constant(C)
+   auto n_pages = UNDO_INITIAL_SIZE_IN_PAGES;      // Use param(P) in normal execution
+   auto space = fil_space_get(old_space_id);       // Get historical workload status(V)
+   if (space->m_undo_extend > UNDO_INITIAL_SIZE_IN_PAGES &&
+       space->m_last_extended.elapsed() < 1000) {
+     n_pages = fil_space_get_size(old_space_id) / 4;
+   }
    fil_ibd_create(…, n_pages);      Calculated from        UNDO tablespace
}                                    historical workload value   extends aggressively
```

Fig. 4: Example of configuration adjustment at runtime based on historical data.

to workload-related variables. However, it falls short in two aspects. First, developers do not comprehensively consider all possible scenarios, adapting partially. Second, developers may fully respect to user-set value and only modify the values when users do not set the parameter, neglecting the necessity of runtime adaption.

Lack of adaption to all scenarios: Developers often use the *max()* or *min()* methods for V→P interaction to select the better value. However, we find that 72.7% (48/66) of these cases only consider one aspect (either the user-set value is too large or too small). For instance, in Figure 2d, the *min()* method prevents the configuration value *threadPoolSize* from exceeding *candidates.size()* (region number) to avoid resource waste. However, if *threadPoolSize* is significantly smaller than the region number, it may lead to performance issues. Conversely, while *max()* can ensure sufficient resources, it may overlook cases where parameters are extensive compared to the required workload, resulting in wasted resources.

Lack of necessary adjustment to user-set values: Sometimes developers are aware of the workload-related variables that will interact with parameters and have taken them into consideration. However, such modification only takes effect when users do not set parameter values at all. Here is an example:

```
1 /* Zookeeper/NIOServerCnxnFactory.java */
2 int numCores = Runtime.getRuntime().availableProcessors();
3 numSelectorThreads =
4   Integer.getInteger(ZOOKEEPER_NIO_NUM_SELECTOR_THREADS,
5     Math.max((int) Math.sqrt((float) numCores / 2) , 1));
```

The value of the selector thread number is relevant to the number of available CPU cores. However, developers use this knowledge only when users do not set parameter *ZOOKEEPER_NIO_NUM_SELECTOR_THREADS*. Even if the user sets an outrageous value, the developer will respect it here. We find 4 such cases in our dataset.

In practice, developers sometimes use historical data to adjust the configuration value, instead of blindly choosing the value set by the user [32]. An example is shown in Figure 4. Developers constantly check workload-related variables (*space->m_undo_exten* and *space->m_last_extended.elapsed()*) to determine whether the UNDO tablespace extends aggressively. If so, developers will use historical data (*fil_space_get_size(old_space_id) / 4*) as the page size rather than statically use user-set value (*UNDO_INITIAL_SIZE_IN_PAGES*).


```
                                /* Yarn/FederationClientInterceptor.java */
int initSize = Math.min(INITIAL_THREAD_POOL_SIZE, maxThreadPoolSize);
threadPool = new ThreadPoolExecutor(initSize, …);    // Initialize with a small constant(C)
while (…) {
  int threadPoolSize = threadPool.getCorePoolSize();
  int nodeNum = allNodes.size();  // Get workload status (V)
  int idealThreadPoolSize = Math.min(maxThreadPoolSize, nodeNum); // Combine P and V
  if (threadPoolSize < idealThreadPoolSize)                       // as threshold
    ... // Extend thread pool          Not enough for runtime workload
}                              Extend thread pool based on P and V
```

Fig. 5: Example of elastic threshold.

> **Finding 9: During V→P, developers conditionally modify user-set values to adapt to runtime workload.** However, the majority (48/66=72.7%) of conditional modifications only consider partial scenarios. Some modifications (4/66) only take effect when users do not set parameter values at all.

Research Direction: Developers need to make user-configured values adapt to the runtime environment and workload. A possible solution is to auto-adjust the user-set configuration value at runtime by using historical data. Such an approach requires systematic modeling and prediction of specific functions.

3) Inappropriate threshold in P→V: In P→V interactions, parameters typically serve as thresholds in conditional branches, which indicate whether the software is running abnormally. The inappropriate threshold can be dangerous. In HBASE-12971, when setting *replication.source.maxretriesmultiplier* to 300, it will lead to a sleep time of more than 24 hours when a socket timeout exception is thrown [11].

Users often find it challenging to set such parameter values as it is difficult to predict runtime workload. For instance, the parameter *thread_stack* in MySQL on a 32-bit platform has a valid range of 128 KB to 4 GB. However, users set it to 192 KB, and the MySQL server encounters a "thread stack overrun" [14]. The user changes it to 256KB and solves the problem.

Figure 5 shows a good practice in Yarn that makes the threshold elastic at runtime. In this case, the threshold of the thread pool (*idealThreadPoolSize*) is not only determined by users. Rather, it uses a user-set threshold (*maxThreadPoolSize*) and workload-related variable (*nodeNum*) to co-determine the threshold. When the user sets *maxThreadPoolSize* too large, it will be restricted by (*nodeNum*). This is actually a case that uses a V→P interaction to constrain P and then uses P as a threshold in another P→V interaction to control software behavior. Such a combination takes the advantage of balancing user intention with actual software requirements.

> **Finding 10: For parameters serving as thresholds in P→V, they themselves also need appropriate limitations.** Arbitrarily setting their values without thorough consideration of the runtime scenario can lead to severe issues.

Research Direction: Automated tools are needed to make good predictions of threshold or make threshold elastic at runtime by using workload information to constrain threshold-type parameters. Such an operation needs the combination of multiple P&V interactions.

4) Missing optimization opportunities in P-V: Besides functionality, P-V interactions are used for optimization tuning. Unlike a simple switch parameter, the workload-related variables in P-V interaction usually represent the requirement of optimization. For example in Figure 2f, when *stageSub-Sections.size()* is positive, which means there are multiple subsections in the corresponding file system image to process, the configuration value *loadInParallel* should be on to execute parallel loading. However, the default value is false, increasing the possibility of missing optimization even when needed.

When developers introduce a new or advanced feature, they often turn it off for the conservative principles [1], [17], [33]. After using it for a long time in production, they may change the default value to true. For example, in HDFS-12603 [13], Developers turn on async edit logging as default after "*running it in production for quite a while with no issues.*"

> **Finding 11:** P-V demonstrates the need for tuning switches. A simple switch-off could lose the opportunity for performance optimization even when needed. Many of the default values (14/21=66.7%) of parameters used for tuning optimization are false.

Research Direction: Future work should monitor the optimization-related variable and design strategies to automatically suggest the scenarios in which the user should turn on the optimization parameter.

VI. THREATS TO VALIDITY

We mainly choose mature data-intensive software systems, which are widely deployed, highly configurable, and well-documented. What's more, they have a rich development and maintenance history (on GitHub and JIRA). Although we choose different types of software, our findings may not apply to all kinds of software. Besides, even random sampling might introduce bias, potentially affecting the results. However, we still believe that our study is representative enough to reveal the advantages and potential problems of the PCV interactions as we did not restrict our research to a certain type of software or a specific value type of configuration parameters. Static analysis tools may miss some propagation paths due to program complexity. To ensure completeness, we manually examine the code to identify PCV interactions. While we can't guarantee capturing all cases, our dataset includes all interaction types with enough instances for a thorough study.

VII. RELATED WORK

Configuration design and comprehension. Some works focus on understanding and improving software configuration design [1], [5], [34]. The work of Zhang et al [1] closely aligns with our study. They study the design and implementation of software configurations at the source code level. Our research is similarly rooted in the source code but diverges by emphasizing the influence of configurations on software behavior. Sayagh et al. [34] summarized common activities in software configuration engineering undertaken by developers through literature reviews and expert interviews. Some research has focused on the design and usage of Software Product Line (SPL) configurations [35], [36], predominantly employed by developers to facilitate collaborative development and testing. Our work focuses on how configuration parameters affect software behavior at the source code level, but not the activities of the configuration management.

Misconfiguration detection and prevention. There has been a substantial body of studies on misconfiguration [3], [6], [37]–[44]. Many works focus on preventing misconfiguration before software execution [4], [7], [18], [23]. There are also works dedicated to detecting configuration issues at runtime [2], [15], [45], [46]. Our work neither focuses on detecting misconfigurations nor on diagnosing software issues caused by misconfigurations. Instead, we concentrate on the interaction of configuration parameters with constants and variables in the source code. We aim to gain a comprehensive and deep understanding of how runtime configuration affects software behavior and use the knowledge to help developers and users design/use configuration better.

VIII. CONCLUSION

This paper presents, to the best of our knowledge, the first comprehensive study on how configurations affect software at runtime by focusing on the interaction between configuration parameters, constants, and variables (PCV Interaction). This interaction reflects how user intentions, developer knowledge, and runtime environment/workload jointly influence software behavior. We analyze 705 configuration parameters from 10 large-scale software systems, revealing that 66.4% of these parameters interact with constants or variables after parsing. We categorize these interactions and examine their runtime effects, highlighting the risks and potential problems associated with specific interaction patterns. Our findings offer new insights and encourage the development of automated techniques to improve both software configuration by users and configuration design by developers.

ACKNOWLEDGMENTS

REFERENCES

[1] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An evolutionary study of configuration design and implementation in cloud systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 188–200.

[2] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and detecting on-the-fly configuration bugs," in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.

[3] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the 23th ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011, pp. 159–172.

[4] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013, pp. 244,259.

[5] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "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.

[6] P. Wallpapers, "Facebook, instagram, whatsapp hit by global outage," https://www.pukmedia.com/EN/Details/69501, 2021.

[7] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "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.

[8] W. Li, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, and X. Liao, "Challenges and opportunities: an in-depth empirical study on configuration error injection testing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 478–490.

[9] "Hbase-21000. default limits for pressureawarecompactionthroughput-controller are too low." https://issues.apache.org/jira/browse/HBASE-21000, 2018.

[10] "Hbase-24544. recommend upping zk jute.maxbuffer in all but minor installs." https://issues.apache.org/jira/browse/HBASE-24544, 2020.

[11] "Hbase-12971. replication stuck due to large default value for replication.source.maxretriesmultiplier." https://issues.apache.org/jira/browse/HBASE-12971, 2015.

[12] "Hbase-19660. up default retries from 10 to 15 and blocking store files limit from 10 to 16," https://issues.apache.org/jira/browse/HBASE-19660, 2017.

[13] "Hdfs-12603. enable async edit logging by default," https://issues.apache.org/jira/browse/HDFS-12603, 2017.

[14] S. Overflow, "Mysql server's thread_stack parameter - what is it? how big should it be?" https://stackoverflow.com/questions/2919558/mysql-servers-thread-stack-parameter-what-is-it-how-big-should-it-be, 2010.

[15] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: A systematic configuration validation framework for cloud services," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–16.

[16] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 687–700.

[17] H. He, Z. Jia, S. Li, E. Xu, T. Yu, Y. Yu, J. Wang, and X. Liao, "Cpdetector: Using configuration-related performance properties to expose performance bugs," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 623–634.

[18] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 735–751.

[19] T. Wang, H. He, X. Liu, S. Li, Z. Jia, Y. Jiang, Q. Liao, and W. Li, "Conftainter: Static taint analysis for configuration options," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, 2023, pp. 1640–1651.

[20] "Cflow," https://github.com/xlab-uiuc/cflow, 2024.

[21] "Jira," https://issues.apache.org/jira/issues/, 2024.

[22] "Github," https://github.com/, 2024.

[23] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, November 2020.

[24] Y. Hu, G. Huang, and P. Huang, "Automated reasoning and detection of specious configuration in large systems with symbolic execution," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 719–734.

[25] H. He, E. Xu, S. Li, Z. Jia, S. Zheng, Y. Yu, J. Ma, and X. Liao, "When database meets new storage devices: Understanding and exposing performance mismatches via configurations," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1712–1725, 2023.

[26] "Bug #33570629 [innodb] mysql server crash," https://github.com/mysql/mysql-server/commit/1b900503e03d, 2021.

[27] J. Li, S. Li, K. Li, F. Luo, H. Yu, S. Li, and X. Li, "Ecfuzz: Effective configuration fuzzing for large-scale systems," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[28] S. Zhou, X. Liu, S. Li, Z. Jia, Y. Zhang, T. Wang, W. Li, and X. Liao, "Confinlog: Leveraging software logs to infer configuration constraints," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 94–105.

[29] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 293–306.

[30] "Bug #21040050 purge thread must exit sooner at server shutdown," https://github.com/mysql/mysql-server/commit/3e8202ff4439, 2015.

[31] "Followup to bug#45225 locking: hang if drop table with no timeout," https://github.com/mysql/mysql-server/commit/dd42aab8406e, 2010.

[32] "Bug#31965404: Tps regression when automatic undo truncate is enabled," https://github.com/mysql/mysql-server/commit/60fd370cae5a, 2020.

[33] H. He, Z. Jia, S. Li, Y. Yu, C. Zhou, Q. Liao, J. Wang, and X. Liao, "Multi-intention-aware configuration selection for performance tuning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1431–1442.

[34] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 646–673, 2018.

[35] L. R. Soares, J. Meinicke, S. Nadi, C. Kästner, and E. S. de Almeida, "Exploring feature interactions without specifications: A controlled experiment," *ACM SIGPLAN Notices*, vol. 53, no. 9, pp. 40–52, 2018.

[36] S. Nadi, "A study of variability spaces in open source software," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1353–1356.

[37] S. Janardhan, "Update about the october 4th outage," https://engineering.fb.com/2021/10/04/networking-traffic/outage/, 2021.

[38] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 1–16.

[39] B. Maurer, "Fail at scale: Reliability in the face of rapid change," *Queue*, vol. 13, no. 8, pp. 30–46, 2015.

[40] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services." in *OSDI*, vol. 4, 2004, pp. 61–76.

[41] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[42] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 193–202.

[43] A. Rabkin and R. H. Katz, "How hadoop clusters break," *IEEE software*, vol. 30, no. 4, pp. 88–94, 2012.

[44] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 328–343.

[45] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011, pp. 28–28.

[46] S. Zhou, Z. Jiang, S. Li, X. Liu, Z. Jia, Y. Zhang, J. Ma, and H. Mi, "Wmwatcher: Preventing workload-related misconfigurations in production environment," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 279–288.