An Evolutionary Study of Configuration Design and Implementation in Cloud Systems

Yuanliang Zhang*†, Haochen He*, Owolabi Legunsen‡, Shanshan Li*, Wei Dong*, Tianyin Xu†
*National University of Defense Technology, Changsha, Hunan, China
†University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
‡Cornell University, Ithaca, NY 14850, USA
{zhangyuanliang13, hehaochen13, shanshanli, wdong}@nudt.edu.cn, legunsen@cornell.edu, tyxu@illinois.edu

***Abstract*—Many techniques were proposed for detecting software misconfigurations in cloud systems and for diagnosing unintended behavior caused by such misconfigurations. Detection and diagnosis are steps in the right direction: misconfigurations cause many costly failures and severe performance issues. But, we argue that continued focus on detection and diagnosis is symptomatic of a more serious problem: configuration design and implementation are not yet first-class software engineering endeavors in cloud systems. Little is known about how and why developers evolve configuration design and implementation, and the challenges that they face in doing so.**

This paper presents a source-code level study of the evolution of configuration design and implementation in cloud systems. Our goal is to understand the rationale and developer practices for revising initial configuration design/implementation decisions, especially in response to consequences of misconfigurations. To this end, we studied 1178 configuration-related commits from a 2.5 year version-control history of four large-scale, actively-maintained open-source cloud systems (HDFS, HBase, Spark, and Cassandra). We derive new insights into the software configuration engineering process. Our results motivate new techniques for *proactively* reducing misconfigurations by improving the configuration design and implementation process in cloud systems. We highlight a number of future research directions.

I. Introduction

Software configuration design and implementation have significant impact on the functionality, reliability, and performance of large-scale cloud systems. The idea behind configuration is to expose *configuration parameters* which enable *deployment*-time system customization. Using different parameter values, system users (e.g., operators, sysadmins, and DevOps engineers) can port a software system to different environments, accommodate different workloads, or satisfy new user requirements. In cloud systems, configuration parameters are changed constantly. For example, at Facebook, configuration changes are committed thousands of times a day, significantly outpacing source-code changes [1].

With the high velocity of configuration changes, misconfigurations (in the form of erroneous parameter values) inevitably become a major cause of system failures, severe service outages, and downtime. For example, misconfigurations were the second largest cause of service-level disruptions in one of Google's main production services [2]. Misconfigurations also contribute to 16% of production incidents at Facebook [1], including the worst-ever outage of Facebook and Instagram that occurred in March 2019 [3]. Similar statistics and incidents were reported in other systems [4]–[11].

Software configurations also impose significant total cost of ownership on software vendors, who need to diagnose user-reported failures or performance issues caused by misconfigurations. Vendors may even have to compensate users, if the failures lead to outages and downtime. Software vendors also need to support and help users with configuration-related questions, e.g., how to find the right parameter(s) and set the right value(s) [12]. Note that system users are often not developers; they may not understand implementation details or they may not be able to debug code [13]–[15].

Unfortunately, configuration design and implementation have been largely overlooked as first-class software engineering endeavors in cloud systems, except for few recent studies (Section VIII). The focus has been on detecting misconfigurations and diagnosing their consequences [13], [16]–[30]. These efforts tremendously improve *system*-level defenses against misconfigurations, but they do not address the fundamental need for better *software* configuration design and implementation. Yet, better configuration design can effectively reduce user difficulties, reduce configuration complexity while maintaining flexibility, and proactively reduce misconfigurations [12], [31]–[33]. Also, better configuration implementation can help detect and correct misconfigurations earlier to prevent failure damage [13], [16].

The understanding of what constitutes software configuration engineering in cloud systems is preliminary in the literature, compared with other aspects of engineering these software systems (e.g., software architecture, modeling, API design, and testing) which are well studied. Meanwhile, we observed that developers struggle to design and implement configurations. For example, we found that developers raise many configuration-related concerns and questions—"*is the configuration helpful?*" (Spark-25676), "*can we reuse an existing parameter?*" (HDFS-13735), "*what is a reasonable default value?*" (HBase-19148). Furthermore, we found that

This is a preprint of the work (with the same title) published at the 43rd International Conference on Software Engineering (ICSE'21). The content should be exactly the same as the conference proceeding version, except that the conference version does not have the replication package due to its page limit. The version is required by ICSE'21 because "accessibility become vital for such a big, distributed and virtual conference"—due to the global COVID-19 pandemic, ICSE'21 will be a virtual conference in zoomsphere.

TABLE I: Summary of our findings on configuration design and implementation, and their implications.

FINDINGS ABOUT CONFIGURATION INTERFACE	IMPLICATIONS
F.1 Software developers often parameterize constant values into configuration parameters. Performance and reliability tuning are common rationales for such parameterization.	**I.1** Configuration auto-tuning techniques that consider reliability and functionality are needed, in addition to performance-only optimization. Timing parameters are an example (critical to both performance and reliability).
F.2 Over 50% of parameterizations were driven by severe consequences of deficiencies in constant values. Unfortunately, use cases that drove the parameterization were often poorly discussed or documented.	**I.2** Techniques for identifying pathological configuration use cases through testing and analysis are desired. Tools that can identify and categorize use cases could help proactively parameterize deficient constants.
F.3 Only 28.1% of default-value changes mentioned systematic testing; 31.3% of default changes chose values that work around reported issues.	**I.3** Many default values in existing software systems may not be optimal. Research on how to better select default values is needed.
FINDINGS ABOUT CONFIGURATION USAGE	**IMPLICATIONS**
F.4 Most configuration-checking code were added as afterthoughts, postmortem to system failures and performance issues in production.	**I.4** Proactive parameter value checking and validation can prevent many severe consequences (but they are still not a common engineering practice).
F.5 Over 50% of checks added as afterthoughts are basic (non-emptiness and value-range checks); other commits invoked checking code earlier.	**I.5** Automated solutions for generating basic checking code and applying them in program early execution phases are useful and feasible.
F.6 Throwing exceptions is common for handling misconfigurations; auto-correction is not, missing opportunities to help users handle errors.	**I.6** Automatically correcting configuration errors is feasible and should be explored in future research.
F.7 Developers often enhance configuration-related log/exception messages by including related parameters and providing guidance.	**I.7** Techniques on automated enhancement of configuration-related log and exception messages to improve misconfiguration diagnosis are needed.
F.8 Reusing existing parameters in different program locations is a common practice. However, parameter reuse leads to various inconsistencies.	**I.8** Tools are needed for identifying and fixing various inconsistencies among configuration parameters and their code implementations.
FINDINGS ABOUT CONFIGURATION DOCUMENTATION	**IMPLICATIONS**
F.9 Inadequate and outdated information are major reasons behind the changes that enhance configuration documents.	**I.9** Enforcing complete, up-to-date documentation of configuration infomation is still a challenge (despite a lot of research effort).
F.10 Configuration use cases, parameter constraints and dependencies between parameters are commonly added to documents.	**I.10** Configuration documentation should be systematically augmented to include critical, user-facing information.

developers frequently revise configuration design/implementation decisions, usually after observing severe consequences (e.g., failures and performance issues) induced by the initial decisions (Sections IV-A1 and IV-A2).

This paper presents a source-code level study of the evolution of configuration design and implementation in cloud systems, towards filling the knowledge gap and better understanding the needs that configuration engineering must meet. Specifically, we study 1178 configuration-related commits spanning 2.5 years (2017.6–2019.12) in four large-scale, widely-used, and actively-maintained open-source cloud systems (HDFS, HBase, Spark, and Cassandra). Each commit that we study is associated with a JIRA/GitHub issue or a Pull Request link which provides more context about the change and the discussions among developers. (Section III describes our methodology for selecting configuration-related commits).

Our goal is to understand current configuration engineering practices, identify developer pain points, and highlight future research opportunities. We focus on analyzing commits that revise or refine initial configuration design or implementation decisions, instead of commits that add or remove parameters as code evolves. These revisions or refinements were driven by consequences of misconfigurations. Our analysis helps to 1) understand the rationale for the changes, 2) learn design lessons and engineering principles, and 3) motivate future automated solutions that can prevent such consequences.

To systematically analyze configuration-related commits, we propose a taxonomy of configuration design and implementation changes in cloud systems along three dimensions: *1) interface:* why and how developers change the configuration interface (parameters, default values and constraints). *2) usage:* how developers change and improve parameter value checking, error-handling, and uses. *3) documentation:* how developers improve configuration documentation.

Note that this paper focuses on cloud systems, instead of desktop software or mobile apps, because misconfiguring cloud systems results in more far-reaching impact. Moreover, we focus on *runtime* configurations [34] whose values can be changed *post-deployment* without re-compiling the software. Runtime configurations fundamentally differ from *compile-time* configurations such as `#ifdef`-based feature flags [35]. But runtime and compile-time configurations have similar problems. So, there are opportunities to extend techniques that solve problems for one to the other.

This paper makes the following contributions:

- ⋆ **Study and Insights.** We study code changes to understand the evolution of configuration design and implementation in cloud systems. We find insights that motivate future research on reducing misconfigurations in these systems.
- ⋆ **Taxonomy.** We develop a taxonomy of cloud system configuration design and implementation evolution.
- ⋆ **Data.** We release our dataset and scripts at "https://github.com/xlab-uiuc/open-cevo" to help followup research (see Appendix B for the replication package).

Table I summarizes our findings and their implications.

II. TAXONOMY

Figure 1 shows the three parts of our taxonomy of cloud-system configuration engineering: *interface*, *usage*, and *documentation*. We focus on aspects that affect how system users interact with configurations, and not on developer-focused aspects like variability and testability. We organize our study along the categories shown in Table II.

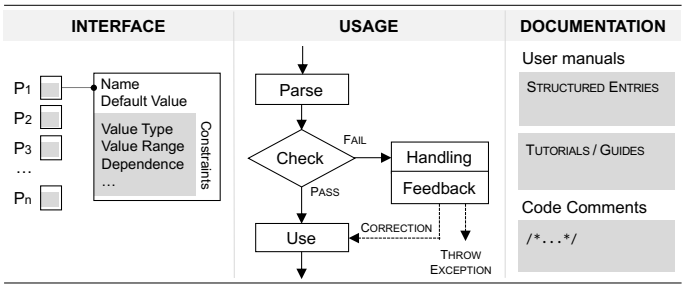

Fig. 1: Three parts of our taxonomy of software configuration design and implementation, and their components.

Interface. The configuration interface that a system exposes to users consists primarily of *configuration parameters* (parameters, for short). As shown in Fig. 1, a parameter is identified by a *name* and it typically has a *default value*. Users can customize system configuration by changing parameter values in configuration files or by using command line interfaces (CLIs). Each parameter places *constraints* (correctness rules) on its values, e.g., type, range, dependency on other parameters. Values that violate the constraints lead to misconfigurations. In Table II, changes that contribute to configuration interface evolution include adding parameters, removing parameters, and modifying parameters.

Usage. Fig. 1 presents the configuration usage model. To use a parameter, the software program first reads its value from a configuration file or CLI, *parses* the value and stores it in a program variable. The variable is then used when the program executes. In principle, the program *checks* the value against the parameter's constraints before *using* it. If checks fail, the program needs to *handle* the error and provide user with *feedback messages*. In Table II, configuration usage evolution consists of changes to all parts of the usage model.

Documentation. These are natural language descriptions related to configurations. We consider changes to user manuals and code comments—the former are written for system users while the latter are written for developers.

III. Study Setup

To understand how configuration design/implementation evolve, we identified and analyzed *configuration-related commits* that modify configuration design and implementation. Following [34], we refer to the design, implementation, and maintenance of software configuration as *configuration engineering*. We start from commits instead of bug databases (e.g., JIRA and GitHub issues) because configuration design and implementation evolution is not limited to bug fixing. All cloud systems that we study record related issue or Pull Request ID(s) in commit messages (Section III-A). We found detailed context about changes in the configuration-related commits through developer discussions. Moreover, commits allow us to analyze the "*diffs*"—the actual changes.

TABLE II: Our taxonomy of configuration engineering evolution.

INTERFACE (SECTION IV)	
AddParam	**Add new configuration parameters**
$\text{Add}_{\text{NewCode}}$	Add new parameters when introducing new modules
$\text{Add}_{\text{CodeChange}}$	Add new parameters due to changes in existing code
$\text{Add}_{\text{Parameterization}}$	Convert constant values to configuration parameters
RemoveParam	**Remove existing configuration parameters**
$\text{Rmv}_{\text{RmvModule}}$	Remove parameters when removing existing modules
$\text{Rmv}_{\text{Replace}}$	Replace parameters with constants or automation
ModifyParam	**Modify existing configuration parameters**
$\text{Mod}_{\text{Naming}}$	Change the name of a configuration parameter
$\text{Mod}_{\text{DefaultValue}}$	Change the default value of a configuration parameter
$\text{Mod}_{\text{Constraint}}$	Change the constraints of a configuration parameter
USAGE (SECTION V)	
Parse	**Change configuration parsing code**
Check	**Change configuration checking code**
Handle	**Change configuration error handling code**
$\text{Handle}_{\text{Action}}$	Handling actions (correction and exceptions)
$\text{Handle}_{\text{Message}}$	Feedback messages (log and exception messages)
Use	**Change how configuration values are used**
$\text{Use}_{\text{Change}}$	Change existing code that uses parameters
Use_{Add}	Add code to reuse a configuration parameter
DOCUMENTATION (SECTION VI)	
User manual	**Change configuration-related user manual content**
Code comment	**Change configuration-related source code comments**

TABLE III: Software and their commits that we study.

SUBJECT	#DESCRIPTION	#PARAMS	#ALLC	#STUDIEDC
HDFS	File system	560	1618	221
HBase	Database	218	3516	268
Spark	Data processing	442	6194	602
Cassandra	Database	220	1868	87

A. Target Software and Version Histories

We study configuration design and implementation in four open-source cloud systems, shown in Table III: HDFS, HBase, Spark, and Cassandra. These projects 1) have many configuration parameters and configuration-related commits, 2) are mature, actively-developed and widely-used, with well-organized GitHub repositories and bug databases, (3) link to issue IDs in commit messages, and 4) are commonly used subjects in cloud and datacenter systems research.

In these subjects, we studied configuration-related commits from June 2017 to December 2019, a 2.5–year time span. In Table III, "**#PARAMS**" is the total number of documented parameters in the most recent version, "**#ALLC**" is the total number of commits in the 2.5–year span, and "**#STUDIEDC**" is the number of configuration-related commits that we studied. We excluded configuration-related commits that only added or modified test cases; we expected such commits to yield less insights on design/implementation evolution. In total, we studied 1178 configuration-related commits.

B. Data Collection and Analysis

To find *configuration-related commits* within our chosen time span, we wrote scripts to automate the analysis of commit messages and diffs, filter out irrelevant commits, and select likely configuration-related commits. Then, we manually inspected each resulting commit and its associated

issue. Overall, we collected 384 commits by analyzing commit messages and 794 commits by analyzing the commit diffs, yielding a total of 1178 configuration-related commits.

1) Analysis of Commit Messages: Keyword search on commit messages is commonly used to find related commits, e.g., [36]–[41]. We manually performed a formative study with hundreds of commit messages and found that three strings commonly occur in configuration-related commits: "config", "parameter" and "option". These strings were previously used in keyword searches [34], [36], and matched 525 times in all four subjects. We manually inspected these 525 commits and removed commits that did not change configurations, yielding 384 configuration-related commits.

2) Analysis of Commit Diffs: Many commit messages do not match during keyword search, even though the diffs show configuration-related changes. So, we further analyzed diffs to find more configuration-related commits, and found additional 794 configuration-related commits. Our diff analysis determines whether diffs modify how parameters are defined, loaded, used, or described. Accurate automated diff analysis requires applying precise taint tracking—treating parameter values as initial taints that are propagated along control- and data-flow paths [13], [16]–[18], [20], [26], [36], [42]–[44]—to each commit and its predecessor and comparing the taint results in *both* commits. Such pairwise analysis does not scale well to the 13196 commits in all four projects (Table III).

To scale diff analysis, we used a simple text-based search of configuration metadata, including the 1) configuration interface (including how configurations are defined and loaded), 2) default configuration file, and 3) message that contains configuration information. Metadata are expected to be stable in the mature cloud systems that we study; commits that modify them may yield good insights on configuration evolution.

Finding commits that change parameter definitions. We start from commits that change default configuration files or parameter descriptions in those files. These two locations are key user-facing parts of configuration design [13], [45]. Thus, modification of parameters (introduction, deprecation, changes to default values, etc.) likely requires changes to either. This heuristic was effective: it found 272 additional configuration-related commits with an average false positive rate of 3.2%.

Finding commits that change parameter loading or setting. Here, we leverage knowledge of configuration APIs. As reported in prior studies [13], [16], [42]–[44], [46] and validated in our study, mature software projects have unified, well-defined APIs for retrieving and assigning parameter values. For instance, HDFS has getter or setter methods (e.g., `getInt`, `getBoolean`, declared in a Java class; each of which has a corresponding setter method (e.g., `setInt`, `setBoolean`). The other evaluation subjects follow this pattern.[1] So, identifying commits that changed code containing getter or setter method usage requires a few lines of code using regular expressions.

[1]This is common in Java and Scala projects: the configuration interface typically wraps around core library APIs such as `java.util.Properties` to provide configuration getter and setter methods.

TABLE IV: Configuration-related commits by category. Some commits contain changes in multiple categories.

	INTERFACE	BEHAVIOR	DOCUMENT	COMMIT
HDFS	139 (62.9%)	58 (26.2%)	27 (12.2%)	221
HBase	171 (63.8%)	87 (32.5%)	21 (7.8%)	268
Spark	367 (61.0%)	182 (30.2%)	61 (10.1%)	602
Cassandra	54 (62.1%)	32 (36.8%)	5 (5.7%)	87
Total	731 (62.1%)	359 (30.5%)	114 (9.7%)	1178

This heuristic found 457 additional configuration-related commits with a 19.9% average false positive rate.

Finding commits that change parameter value data flow. If a commit changes code with variables that store parameter values, then that commit is likely related to the data flow of parameter values. We implemented a simple text-based taint tracking to track such variables as follows. Once a configuration value is stored in a variable, we add the variable name to a global taint set. We perform the tracking for every commit in the time span that we studied. We do not remove variables from our taint set. We output candidate commits where a modified statement contains a variable name in the taint set. Taint tracking found 31 additional configuration-related commits with an average false positive rate of 26.2%.

Identifying other configuration-related commits We applied the same keyword search on commit messages (Section III-B1) to messages that occur in diffs, to capture commits that change related exception or log messages without modifying any other code. We found 34 additional configuration-related commits with an average false positive rate of 29.2%.

3) Inspection and Categorization: At least two authors independently studied each configuration-related commit and its corresponding issue. They independently categorized each commit based on the taxonomy in Section II, and then met to compare their categorization. When they diverged, a third author was consulted for additional discussion until consensus was reached. Further, in twice-weekly project meetings, the inspectors met with a fourth author to review their categorization of 15% of commits inspected during the week. These meetings helped check that understanding of the taxonomy is consistent. Our experience shows that consistently checking a taxonomy like Figure 1 with concrete examples significantly improves inter-rater reliability and categorization efficiency.

Note that we categorized each commit based on how it revised the original configuration design/implementation. If a commit adds a new parameter and also a manual entry to document this new parameter, we treat this commit as $\text{Add}_{\text{Param}}$ (Table II)—the commit revises the configuration interface instead of documentation. Some commits modify multiple (sub-)parts in our taxonomy.

4) Data Collection Results: Table IV shows the studied configuration-related commits along the three parts of our taxonomy. There is a significant number of commits in each part. The rest of this paper summarizes our analysis and provides insights on how configuration design and implementation evolve along these three parts.

IV. Configuration Interface Evolution

Changes to the configuration *interface* were the most common, compared with behavior or documentation changes (Table IV). We focus on analyzing changes to *configurability*—the level of user-facing configuration flexibility—(Section IV-A) and default values (Section IV-B). We omit other kinds of configuration interface changes which are often routine and cannot directly lead to misconfigurations.

A. Evolution of User-Facing Configuration

Table V shows our categorization of changes to configurability. Most changes add or remove parameters; per project, removal is 5.1× to 21.2× less frequent than addition (with an average of 8.4×). We find that adding or removing parameters occur naturally during software evolution—parameters are added with new code, and removed with code deletion. We do not focus on co-addition or co-removal of parameters with code. Rather, we focus on changes that revise previous configuration engineering decisions by 1) parameterizing constants and 2) eliminating parameters or converting them to constants.

Our data corroborates a prior finding [12] that configuration interface complexity increases rapidly over time, as more parameters are added than are removed. Complexity measures the size of the configuration space (number of parameters multiplied by the number of all their possible values). Approaches for dealing with the rapid growth rate are desired. Variability modeling [47]–[51] which is extensively researched for compile-time configurations, can potentially be extended to understand and manage runtime configuration complexity.

1) Parameterization: Developers often convert constants into parameters after discovering that one constant cannot satisfy all use cases. We find 142 commits that parameterize 169 constants (169 parameterizations). We report on 1) rationales for the parameterizations, 2) how developers identify constants to parameterize, 3) use cases that made constants insufficient, and 4) how developers balance increase in configuration complexity (caused by adding new parameters) with the need for flexibility (which necessitates parameterization). Our results have ramifications for configuration interface design: we provide understanding for managing the configurability versus simplicity tradeoff. The rationales for parameterization also motivate configuration parameter auto-tuning.

Rationales for parameterization. These include: performance tuning, reliability, environment setup, manageability, debugging, compatibility, testability, and security. Table VI shows, for each rationale, the number of commits and parameters, an example parameter, and a description. We discuss the top two rationales, due to space limits. Performance tuning was the top rationale for parameterizing constants, involving 39.6% (67/169) of parameters in 56 commits. Different workloads need different values, so it is hard to find one-size-fits-all constants. Resource-related (e.g., buffer size and thread number), feature selection (turning on/off features with performance impact, e.g., monitoring), and timing logic (mostly timeouts and intervals) were the main resulting parameter types, with 20.9% (14/67), 37.3% (25/67), and 14.9% (10/67) new parameters, respectively. Others 26.9% (18/67) set algorithm-specific parameters (e.g., weights and sample sizes).

TABLE V: Statistics on configuration interface changes.

	HDFS	HBase	Spark	Cassandra	Total
AddParam	106	122	277	42	547
$\text{Add}_{\text{NewCode}}$	54	55	143	23	275
$\text{Add}_{\text{CodeChange}}$	16	34	72	8	130
$\text{Add}_{\text{Parameterization}}$	36	33	62	11	142
RemoveParam	5	24	30	6	65
$\text{Rmv}_{\text{RmvModule}}$	3	16	25	4	48
$\text{Rmv}_{\text{Replace}}$	2	8	5	2	17
ModifyParam	28	25	60	6	119
$\text{Mod}_{\text{Naming}}$	5	8	30	1	44
$\text{Mod}_{\text{DefaultValue}}$	19	14	20	3	56
$\text{Mod}_{\text{Constraint}}$	4	3	10	2	19

Reliability, with 37 of 169 of the parameterizations, was the second most common rationale. Of these, 17 were caused by hardcoded timeout values that led to constant request failures in the reported deployments, so developers made them configurable. Note that new timing parameters were created for both reliability and performance tuning. For example, in HDFS, a new timing parameter was created to improve performance. The previous constant was causing a "*delete file task to wait for... too long*" (HBase-20401). But, another HDFS timing parameter was created to improve reliability. The previous constant was too small, causing "*timeouts while creating 3TB volume*" (HDFS-12210).

> **Discussion:** Configuration auto-tuning techniques that consider reliability and functionality are needed, in addition to performance-only optimization [52]–[70]. Specifically, timing parameters have important implications to both reliability and performance; however, not much work has been done on auto-tuning timing parameters (e.g., timeouts and intervals).

How developers find constants to parameterize. 54.4% (92/169) of parameterizations were *postmortem* to severe consequences, e.g., system failures, performance degradation, resource overuse, and incorrect results. Among previous constants for these, 40.2% (37/92) led to performance degradation; 35.9% (33/92) caused severe failures; 19.6% (18/92) led to incorrect or unexpected results (e.g., data loss and wrong output); and 4.3% (4/92) resulted in resource overuse.

> **Discussion:** Despite the efforts in parameterization, developers still overlook deficient constants that may lead to severe consequences (e.g., failures and performance issues). *Proactive* techniques for detecting deficient constants and for automating parameterization are needed; the latter could assist performance testing of cloud systems.

TABLE VI: Statistics and examples of developers' rationales for parameterization (excluding two commits that lacks information).

RATIONALE	#COMMIT	#PARAM	EXAMPLE NEW PARAMETER	LIMITATION OF PREVIOUS CONSTANT
Performance	56	67	spark.sql.codegen.cache.maxEntries	The cache size does not work for online stream processing (Spark-24727)
Reliability	28	37	spark.sql.broadcastExchange.maxThreadThreshold	Out of memory if thread-object garbage collection is too slow (Spark-26601)
Manageability	20	20	dfs.federation.router.default.nameservice.enable	Enable the default name service to store files (HDFS-13857)
Debugging	8	9	spark.kubernetes.deleteExecutors	Disable auto-deletion of pods for debugging and diagnosis (Spark-25515)
Environment	8	13	dfs.cblock.iscsi.advertised.ip	Allows server and target addresses to be different (HDFS-13018)
Compatibility	13	13	spark.network.remoteReadNioBufferConversion	Add the parameter to fall back to an old code path (Spark-24307)
Testability	3	4	spark.security.credentials.renewalRatio	May not need to be set in production but can make testing easier (Spark-23361)
Security	4	4	spark.sql.redaction.string.regex	The output of query explanation can contain sensitive information (Spark-22791)

TABLE VII: Use-case description of parameterization changes.

LEVEL	EXAMPLE
Concrete	"*Volume creation times out while creating **3TB** volume*" (HDFS-12210).
Vague	"*If **many** regions on a RegionServer, the default will be not enough*" (HBase-21764).
No Info	"*It **would be better** if the user has the option instead of a constant*" (Spark-25233).

Describing use cases that prompt parameterization. Use cases where constants were deficient should be described fully to help users set correct values. But, developers described the *concrete* use cases that prompt parameterization for only 37.9% (64/169) parameters. Others discussed use cases either *vaguely* (45.0% or 76/169 parameters) or provided *no information* (17.1% or 29/169 parameters). Table VII shows examples.

DISCUSSION: Future work should identify and document use cases and workloads, including which parameters can be tuned, and suggest beneficial configuration values that are designed for concrete use cases.

Balancing flexibility and simplicity Configuration interface design must balance flexibility (i.e., configurability) with simplicity [12]. New parameters increase flexibility (by handling additional use cases), but increase interface complexity (thus reducing usability). 16.2% (23/142) of parameterization commits contained developer discussions on the flexibility-simplicity tradeoff. Most discussed estimated prevalence of use cases for parameterization—it is not worth increasing interface complexity for rare use cases—and typically involve advanced users, e.g., "*admittedly, this...is an expert-level setting, useful in some cases*" (Cassandra-14580). We also found developers' debates on whether to parameterize (Cassandra-12526, HDFS-12496, Spark-26118).

A *middle-ground* solution is to parameterize without documenting or exposing the parameter, e.g., "*although...not widely used, I could see allowing control...via an undocumented parameter*" (Spark-23820). With this practice, not all but the most advanced users know of such parameters. We found that 58.0% (98/169) of the newly added parameters were not documented in the parameterization commit, indicating that these parameters were first added as middle-ground solutions.

DISCUSSION: Further studies are needed on 1) if and why undocumented parameters are eventually documented, and 2) how often and why (expert) users modify un-exposed parameters, in order to understand the intent and utility of such parameters.

Specifically, visibility conditions from variability modeling [49], [50], [71] can be extended to manage the tradeoff of flexibility versus simplicity, which can benefit navigation support and user guidance [72]–[75]. Currently, visibility conditions are mainly designed for Boolean feature flags based on dependency specifications (e.g., in CDL and KConfig); complexity metrics and variability analysis for other parameter types (e.g., numeric and strings) are needed.

2) Removing Parameters: Understanding parameter removal can yield insights on reducing configuration interface complexity [12], [35]. We examined all 17 configuration-related commits that removed a parameter (not co-removal with code). All 17 removed parameters were converted to constants or code logic was added that obviated them. 14 removed parameters were converted to constants: 11 to their default values and 3 to safe values. Developers mentioned that 8 of the 17 parameters had no clear use case (e.g., HBase-8518, HBase-18786), or required users to understand implementation details (e.g., Cassandra-14108). Three parameters confused users or might lead to severe errors (e.g., Spark-26362).

Three of 17 removed parameters were obviated by new automation logic. For example, in HBase-21228, `hbase.regionserver.handler.count` which specified the number of concurrently updating threads to be garbage collected in a Java `ConcurrentHashMap`, was removed after developers switched to `ThreadLocal<SyncFuture>` which automatically garbage collects terminated threads. This example shows how implementation choices could affect configuration complexity.

DISCUSSION: Future studies can evaluate the utility and impact of each parameter (e.g., by analyzing if and how often deployed values are equal or similar to the default values). Configurations with low utility can be replaced with constants (e.g., default values).

B. Evolution of Default Values

Parameter default values are important to the usability of configurable systems; they provide users with good starting points for setting parameters without needing to understand the entire configuration space. Thus, developers usually choose default values that satisfy common use cases. Ideally, a default value applies under most common workloads, without causing failures (HBase-16417, HBase-20390, HDFS-11998). The "$\text{Mod}_{\text{DefaultValue}}$" row in Table V shows 56 commits that changed 81 default values. We discuss why default values changed and how new default values were chosen.

Reasons for changing default values. We observe proactive and reactive default value changes. 38.3% (31/81) default-value changes were proactive, including 1) enabling a previously disabled feature flag (32.3% (10/31)), e.g., "*running the feature in production for a while with no issues, so enabled the feature by default*" (HDFS-7964), 2) performance reasons (35.4%, 11/31), e.g., "*sets properties at values yielding optimal performance*" (HBase-16417), and 3) supporting new use cases (32.3%, 10/31), e.g., *"it may be a common use case to ...list queries on these values*" (Cassandra-14498).

The remaining 61.7% (50/81) of default value changes were reactive to user-reported issues, including 1) system failures and performance anomalies due to not supporting new workloads, deployment scale, hardware, etc (50.0%, 25/50), 2) inconsistencies with user manual (38.0%, 19/50), and 3) working around software bugs (12.0%, 6/50), e.g., *"we set the parameter to false by default for Spark 2.3 and re-enable it after addressing the lock congestion issue"* (Spark-23310).

Choosing new values. It is straightforward to change new default values for Boolean and enumerative parameters, given their small value ranges. So, we describe how new default values of 32 *numeric* parameters were chosen (excluding those that fix default value inconsistency (e.g., HBase-18662). Only 28.1% (9/32) numeric parameters had systematic performance testing and benchmarking mentioned in the JIRA/GitHub issues. Later commits reset these new default values, despite the initial testing and benchmarking. For example, HBase developers performed "*write-only workload evaluation...read performance in read-write workloads. We investigate several settings...*" (HBase-16417). Yet, we found three later commits that changed the default value of the same parameter to different numbers. For 31.3% (10/32) of numeric parameters, new default values were chosen by adjusting the previous default values to resolve production failures. In many of these cases, usually without high confidence in the new default values, developers simply chose values that resolve the problem(s). Examples: "*It probably makes sense to set it to something lower*" (Spark-24297), or "*I'm thinking something like 3000 or 5000 would be safer*" (HBase-18023). We found no information on the remaining 40.6% (13/32) numeric parameters.

We observe that backward compatibility and safety are common considerations in selecting new default values. New default values that radically change system behavior are often considered inappropriate (e.g., HBase-18662).

TABLE VIII: Statistics on configuration usage evolution.

	HDFS	HBase	Spark	Cassandra	total
Parse	5	14	59	7	85
Check	7	20	29	11	67
Handle	12	18	20	2	52
$\text{Handle}_{\text{Action}}$	8	6	4	1	19
$\text{Handle}_{\text{Message}}$	4	12	16	1	33
Use	34	35	74	12	155
$\text{Use}_{\text{Change}}$	7	10	25	3	45
Use_{Add}	27	25	49	9	110

Discussion: Default value changes are often reactive to the reported issues, without systematic assessment. Systematic testing and evaluation of new (and existing) default values are needed.

Dynamic workloads and heterogenous deployments necessitate continuous and incremental changes to default values. Future work could maintain a set of default values (instead of one) for typical workloads, hardware, and scale.

C. Summary

There is an unmet need for *practical* configuration automation techniques and tools for choosing and testing parameter values—why do cloud system developers still change parameter values statically rather than using parameter automation? There is also need for automatic ways of identifying workloads or use cases for which default values (and even constants) are ill-suited. Such automatic workload identification approaches can help developers to better 1) decide which constant values need to be parameterized, 2) understand when their current default values will lead to system failures, and 3) come up with better tests and benchmarks for default values.

V. Configuration Usage Evolution

We present results on configuration usage evolution (recall the configuration usage model described in Fig. 1 and Section II). Across the four cloud systems, 26.2%–36.8% of commits changed parameter usage (Table VIII). We describe changes to checking, error handing, and use of parameters. We omit changes to parsing APIs (e.g., Spark-23207).

A. Evolution of Parameter Checking Code

Proactively checking parameter values is key to preventing misconfigurations [16]. However, we find that many parameters had no *checking code* when they were introduced. Checking code was added *reactively*: 1) 74.6% (50/67) of commits that changed checking code occurred after users reported runtime failures, service unavailability, incorrect/unexpected results, startup failures, etc. (Table IX shows examples). 2) In 14.9% (10/67) commits, developers *proactively* added or improved the checking code; 2 of them applied reactively-added checking code to other parameters with similar types (e.g., Cassandra-13622). 3) We did not find sufficient information of the other 7 commits.

TABLE IX: Examples of consequences that can be prevented by adding configuration checking code.

CONSEQUENCE	EXAMPLE PARAMETER	DESCRIPTION
Runtime Error	hbase.bucketcache.bucket.sizes	If value is not aligned with 256, instantiating a bucket cache throws IOException (HBase-16993)
Early Termination	commitlog_segment_size_in_mb	If value $\geq$ 2048, Cassandra throws an exception when creating commit logs (Cassandra-13565)
Service unavailability	spark.dynamicAllocation.enabled	Running barrier stage with dynamic resource allocation may cause deadlocks (Spark-24954)
Unexpected Results	spark.sql.shuffle.partitions	If the value is 0, the result of a table join will be an empty table (Spark-24783)

```
+ if (writeTables==null || writeTables.isEmpty()) {
+   throw new IllegalArgumentException(
+     "Configurtion parameter" +
+     OUTPUT_TABLE_NAME_CONF_KEY + "cannot be empty")}
```

(a) Add a NOT-NULL check (HBase-18161)

```
+  if (bucketSize % 256 != 0) {
+    throw new IllegalArgumentException(
+      "Illegal value:" + bucketSize +
+      "configured for" + BUCKET_CACHE_BUCKETS_KEY +
+      "All bucket sizes to be multiples of 256")}
```

(b) Add a semantic check (HBase-16993)

```
+  require(conf.getOption(authKey).isEmpty()
+    || !restServerEnabled,
+     s"The RestSubmissionServer does not " +
+      "support authentication via ${authKey}." +
+      "Either turn off spark.master.rest.enabled " +
+      "or do not use authentication.")
```

(c) Add a check for parameters dependency (Spark-25088)

```
+  if (rdd.isBarrier() &&
+      Utils.isDynamicAllocationEnabled(sc.getConf)) {
+    throw new SparkException(
+      "Barrier execution mode does not support"
+      "dynamic resource allocation for now. You can"
+      "disable dynamic resource allocation: setting"
+      "spark.dynamicAllocation.enabled to false")}
```

(d) Add a check for execution context (Spark-24954)

Fig. 2: Examples of configuration checking code.

1) Adding new checking code: 79 new checks were added in 83.6% (56/67) of checking-code related commits. 87.3% (69/79) of these new checks were for specific parameters, while the others were applied to groups of configuration parameters (e.g., read-only parameters). Surprisingly, for specific parameter checks (69 checks in 46 commits), 58.0% (40/69) were basic checks: NOT-NULL (20/69), value range (15/69) and deprecation checks (5/69). An example is in Fig. 2(a). Majority of new checking code were added reactively, corroborating that simple checks can prevent many severe failures [16], [76]. More of such checks could be automatically added and invoked at system startup. The other 29 checks were more complex: 9 value semantic checks (e.g., file/URI properties and data alignment, Fig. 2(b)), 2) 13 checks for parameter dependencies (e.g., Fig. 2(c)), and 3) 7 checks for execution context (e.g., Fig. 2(d)).

2) Improving existing checking code: 11 commits improved existing checking code: eight made checks more strict, e.g., a NOT-NULL check was improved to "*only allow table replication for sync replication*" (HBase-19935), and three moved checking code to be invoked earlier instead of during subsequent execution, e.g., "*when starting task scheduler, spark.task.cpus should be checked*" (Spark-27192).

DISCUSSION: Checks for parameter values are often added as afterthoughts. Proactively generating checking code can help prevent failures due to misconfigurations.

Two possible directions are automatically learning checking code (we find that newly-added checking code is often simple) and automatically applying checking code for one parameter to other parameters both in the same software (which developers are already doing manually) and across software projects. A direction is to co-learning checking code and usage code. Techniques for extracting complex constraints and specifications can reduce manual effort for reasoning about and implementing checking code. A few recent works show promise for inferring parameter constraints through analysis of source code and documentation [13], [77]–[79]. Techniques for extracting feature constraints could be extended and applied to runtime configurations [80]–[83].

B. Evolution of Error-Handling Code

We discuss changes to misconfiguration-related exception-handling code and to messages that provide user feedback.

1) Changes to configuration error handling: 19 commits dealt with error handling: 10 added new handling code to try-catch blocks or throw new exceptions; 9 commits changed handling code. Among the 9 commits, (1) four changed misconfiguration-correction code: three of these added logic to handle a misconfiguration, e.g. "*if secret file specified in httpfs.authentication.signature.secret.file does not exist, random secret is generated*" (HDFS-13654) and one changed buggy misconfiguration-correction code to simply log errors (HDFS-14193) (showing that auto-correcting misconfigurations is not always easy), (2) three changed the exception type as it was "*dangerous to throw statements whose exception class does not accurately describe why they are thrown...since it makes correctly handling them challenging*" (HDFS-14486), and (3) two replaced exception throwing with logging the errors and resuming the execution.

We also studied the newly added handling code in the 79 commits that added new checking code in Section V-A1. In 73.4% (58/79) of the cases, the handling code threw runtime exceptions or logged error messages. The expectation is that users should handle the errors. In the remaining 26.6% (21/79) cases, developers attempted to correct the misconfigurations,

TABLE X: Four levels of message feedback quality in commits that changed exception or logging messages.

LEVEL	DESCRIPTION	EXAMPLE
L4	Contain parameter names and provide guidance for fixing	"Barrier execution mode does not support dynamic resource allocation... You can disable dynamic resource allocation by setting...`spark.dynamicAllocation.enabled` to false." (Spark-24954)
L3	Contain parameter names	"Failed to create SSL context using `server_encryption_options`." (Cassandra-14991)
L2	Do not contain parameter names	"This is commonly a result of insufficient YARN configuration." (HBase-18679)
L1	No mention of configuration	"Could not modify concurrent moves thread count." (HDFS-14258)

e.g., "*it's developers' responsibility to make sure the configuration don't break code.*" (Spark-24610). Developers corrected misconfigurations by changing to the closest value in the valid range (11/21), reverting to the default value (3/21), and using the value of another parameter with similar semantics (7/21).

DISCUSSION: Developers want to make code more robust in the presence of misconfigurations, but their manual efforts are often ad hoc. There is need for new techniques for generating misconfiguration correction code and improving existing handling code.

Techniques for fixing compile-time configuration errors, such as range fixes [84], [85], may be applicable for generating correction strategies for some types of runtime parameters. A key challenge is to attribute runtime errors (e.g., exceptions) to misconfigurations and to rerun the related execution with the corrected configurations.

2) Changes to feedback messages: Feedback (error log or exception) messages are important for users to diagnose and repair misconfigurations. We investigated commits that modified feedback messages and categorize the level of feedback that they provided in Table X, where L4 messages provide the highest-quality feedback and L1 messages provide the lowest-quality feedback. Among 33 commits that modified messages, 18 enhanced feedback quality by adding configuration-specific information. After enhancement, 8 messages became `L3`, and 7 became L4. Changes in the other 15 commits improved 1) correctness (9/15)—half changed imprecise parameter boundary values, e.g., from "no less" to "greater" (Spark-26564), 2) readability (3/15), such as fixing typographic errors, 3) the log level (2/15), and 4) security (1/15), i.e., removing potentially sensitive value.

DISCUSSION: Future work could study the feedback level in *all* messages related to misconfiguration handling code. If most messages are not L4, then future work should automatically detect deficient messages and automatically enhance them to L4.

Moreover, configuration-related logging is not as mature as logging for debugging [86]–[91]. Improving configuration-related logging requires logging related parameters, erroneous values, and, where feasible, possible fixes. Poor-quality feedback from tools hinders developers [92] and techniques exist for dealing with message errors in other domains [93], [94].

C. Evolution of Parameter Value Usage

Software developers change how existing parameters are used ("$\text{Use}_{\text{Change}}$" in Table VIII) and reuse existing parameters for different purposes ("Use_{Add}" in Table VIII).

1) Changing how existing parameters are used: 45 commits changed parameter usage for the following reasons:

Fine-grained control. In 12/45 commits, developers previously used one parameter for multiple purposes, due to poor design—"e.g., *CompactionChecker and PeriodicMemStoreFlusher execution period are bound together*" (HBase-22596)—or for reuse—e.g., "*arrow.enabled was added... with PySpark... Later, SparkR... was added... controlled by the same parameter. Suppose users want to share some JVM between PySpark and SparkR... They use the optimization for all or none.*" (Spark-27834). Developers resolved both categories by creating separate parameters for fine-grained control.

Domain/scope. 8/45 commits changed the usage domain or scope of a parameter. For example, HDFS developers changed a parameter, which was previously only used in the decommission phase to also be used in the maintenance phase, so "*lots of code can be shared*" (HDFS-9388).

Parameter overriding 9/45 commits changed parameter override priority, e.g., "*We need to support both table-level parameters. Users might also use session-level parameter... the precedence would be...*" (Spark-21786).

Semantics 6/45 commits changed what a parameter is used for, e.g., in Spark-21871, developers started using `spark.sql.codegen.hugeMethodLimit` as the maximum compiled function size instead of `spark.sql.codegen.maxLinesPerFunction`.

Parameter replacement 6/45 commits swapped one parameter for another because the previous one was outdated or wrong , e.g., in Spark-24367, a use of `parquet.enable.summary-metadata` was replaced with a use of `parquet.summary.metadata.level` because the former was deprecated.

Buggy parameter values 4/45 commits changed parameter values that were buggy, e.g., the value of a parameter changed because, "*user specified filters are not applied in YARN mode...we need... user provided filters*" (Spark-26255).

2) Reusing existing parameters: To avoid growing the configuration space unnecessarily, developers sometimes reuse existing parameters that are similar to their new use case, instead of introducing a new parameter. 110 commits reused 151 existing parameters for different purposes. However, parameter

```
+  String principal = conf.get(
+   Constants.REST_KERBEROS_PRINCIPAL);
+   if (principal != null) {...}

Preconditions.checkArgument(principalConfig != null
  && !principalConfig.isEmpty(),
  REST_KERBEROS_PRINCIPAL +
  " should be set if security is enabled");
```

(a) Inconsistent checking (HBase-20590)

```
+  if(peerConf.get("hbase.security.authentication")
+      .equals("kerberos")) {...}

isSecurityEnabled = "kerberos".equalsIgnoreCase(
    conf.get("hbase.security.authentication"));
if (isSecurityEnabled) {...}
```

(b) Inconsistent parameter usage (HBase-20586)

Fig. 3: Examples of configuration inconsistent reuse.

reuse comes at a cost. We find two main problems. First, reusing a parameter and code that it controls can result in subtle inconsistencies that can lead to bugs or user confusion. 19.2% (29/151) parameter reuses had such inconsistencies. Second, developers often clone existing code to enable reuse. We focus on inconsistencies. Problems of code cloning are the subjects of other research.

We manually checked for inconsistencies by comparing the newly-added code in a target commit with code that used the parameter in existing code base. We found 29 inconsistencies in HDFS (9/29), HBase (9/29) and Spark (11/29). Inconsistencies manifest in different ways. We categorized them based on the sources of inconsistencies during reuse: 1) feedback message (9/29), e.g., Spark-18061; 2) checking code (4/29) e.g., HBase-20590; 3) new uses of deprecated parameters (6/29), e.g., HDFS-12895; 4) default values (3/29), e.g., HBase-21809; and 5) use statements (7/29), e.g., HBase-20586. Fig. 3 shows examples of inconsistencies in reuse of checking code and use statements, where added lines start with +. In Fig. 3(a), the new parameter usage did not check for parameter value emptiness as the old usage did. In Fig. 3(b), the new usage of `hbase.security.authentication` checked case-insensitive equality; the old usage was case-sensitive.

DISCUSSION: Inconsistencies in parameter usage can confuse users (the same values are used in different ways) or lead to bugs. Ideas for detecting bugs as deviations from similar program behavior [95], [96] could be starting points for addressing this problem.

D. Summary

We advocate that improving software qualities—resilience, diagnosability, and consistency—should be first-class principles in software configuration engineering. We find that even in mature, production-quality cloud systems, checking, error handling, feedback, and parameter usage are often not designed or implemented in a principled manner. More research effort should be put on enhancing these essential qualities of configurable software to defend against misconfigurations, in addition to detection and diagnosis tools that are external to the cloud system [1], [17]–[28], [30].

VI. Configuration Document Evolution

We very briefly discuss configuration document evolution: 114 commits made 149 changes to user manuals or code comments. 100 of these commits changed user manuals and the rest changed code comments. We discuss why configuration documents were changed and the changed content.

Reasons for changing configuration documents. The 149 changes to configuration documents resolved five types of problems: 1) 63 were inadequate for users to understand parameters or to set values correctly, e.g., "*users wondered why spark.sql.shuffle.partitions...unchanged when they changed the config...worth to explain it in guide doc*" (Spark-25245); 2) 29 were outdated after configuration design and implementation changed (Section IV and Section V); 3) 21 were incorrect, e.g., "*LazyPersistFileScrubber will be disabled if... configured to zero. But the document was incorrect*" (HDFS-12987); 4) 17 had readability issues, e.g., "*Client rpc timeouts are not easy to understand from documentation*" (HBase-21727); and 5) 19 improved content, e.g., "*Add thrift scheduling... config to scheduling docs*" (Spark-20220).

DISCUSSION: Document-as-code techniques can be applied to eliminate inconsistencies between configuration documents and configuration design/implementation.

Content added to enhance documents. Inadequate information was the most common problem resolved by configuration document changes. We put the 63 changes that enhanced inadequate documents in six categories based on the content added: 1) 16 changed constraints on parameter values, e.g., "*This should be positive and less than 2048*" (Cassandra-13622); 2) 10 explained dependence on other parameters, e.g., "*This property works with dfs.namenode.invalidate.work.pct.per.iteration*" (HDFS-12079); 3) 6 changed parameter value types and units; 4) 6 changed parameter scope, e.g., "*Timeout... is controlled differently. Use hbase.client.scanner.timeout.period property to set this timeout*" (HBase-21727), 5) 22 provided use cases and guidance, e.g., "*enabling this will be very helpful if dfs image is large*" (HDFS-13884); and 6) 3 warned about deprecation, e.g., "*this config will be removed in Spark 3.0*" (Spark-25384).

DISCUSSION: Ethnographic studies could help understand the gaps between documented configuration information and configuration obstacles faced by users.

Summary Correctness and effectiveness of technical documentation is a long-lasting problem in software engineering. Configuration documentation is no exception. Specialized techniques for maintaining and improving configuration

documentation are needed. For example, checking for inconsistencies between documents and source code [97]–[100] could help detect defects in configuration-related code or documents. Also, techniques for auto-generating documents, especially using structured data, can be applied to generating per-parameter comments and manual entries [101]–[103].

VII. Threats to Validity

We studied cloud systems. Some of our findings may not generalize to other kinds of software. We chose these projects because they are widely used, highly configurable with lots of parameters, mature, and well maintained. They also have issue-tracking systems that help us understand the context of configuration-related commits.

Though we selected candidate commits from version control history, we may have missed some configuration-related commits due to two limitations. First, our regular expressions assume standard coding conventions and will not match if developers do not follow these conventions. Second, our simple, text-based tainting may miss some changes to the data flow of variables that store parameter values. However, as we mentioned in Section III, precise pairwise tainting does not scale to all the commits in the range that we studied—we traded off precision for scalability. All commits selected were manually inspected and categorized through a rigorous quality-assurance process (Section III-B3). False positives came mainly from commits that touched lines containing configuration-related variables but did not change the configuration.

VIII. Related Work

A prior study [36] found that software evolution necessitates resetting parameter values and built `ConfigSuggester` to identify parameters whose values need to be changed after a software updates. We study how the configuration interface and parameter usage change across (a portion of) version control history to draw insights for better configuration design and implementation. Sayagh et al. [34] studied software configuration engineering in practice using interviews, user surveys, and a literature review. Our work is complementary: we perform a *code*-level study of configuration evolution, which yields new insights.

There have been many studies on misconfigurations in a wide variety of software systems [1], [2], [4]–[11]. Our work does not focus on detecting misconfigurations or diagnosing failures caused by misconfigurations. We focus on current configuration engineering practices, with the goal to understand how to improve configuration design and implementation.

Recently, a few studies investigated automated techniques or engineering practices to enhance configuration checking code [16], diagnosability [104], interface [45], security [14], [105], [106], configuration data analysis [107], configuration libraries [108], [109], and correlations or coupling in configuration and code [110]–[112]. Our work corroborates and complements the aforementioned work from the perspective of software evolution. Specifically, our work studies the practices of *software developers* and reveals how software configuration design and implementation are revised and evolved.

Despite the differences (Appendix A), runtime configurations share commonalities with compile-time configurations or SPL configurations, such as `#ifdef`-based feature flags [35]. It is possible that techniques and methodologies designed for compile-time configurations, especially feature and variability modeling [47], [49]–[51], [71], [113], could be adapted for use with runtime configurations. Such adaptation needs to address unique challenges of runtime configuration parameters, such as dependencies on deployment environments, as well as their complex data types and misconfiguration patterns.

Configuration design and implementation have significant implications on software testing and debugging [114]–[119]. For example, introducing new parameters enlarges the configuration space and thus makes it more costly to comprehensively test software. In this paper, we focus on understanding how to improve configuration design and implementation so that fewer misconfigurations occur, and not on software bugs that can occur under different parameter value combinations.

IX. Conclusions

We presented present an evolutionary study of configuration design and implementation in cloud systems. To the best of our knowledge, ours is the first evolutionary study on code-level runtime configuration design and implementation in these systems. We analyze rationales and practices for revising configuration design and implementation decisions, especially in response to consequences of misconfigurations. Our study yields several new insights into the configuration engineering process, and research opportunities for reducing misconfigurations. Our hope is to inspire researchers and developers to treat configuration engineering as a first-class software engineering endeavor.

Appendix A: Runtime versus SPL Configuration

A very frequent request is to compare runtime configuration (the type of configuration studied in this paper) with software product lines (SPL) configuration (often referred to as "feature flags" or "feature toggles") and to position the work in the area of SPL and variability modeling. We explicitly discuss a few fundamental differences:

First, runtime configurations are changed by software users (operators/sysadmins in our context); SPL configurations are managed by developers. Since users are unfamiliar with code, the configuration specifications become the interfaces (Section IV). Moreover, as users are prone to misconfigurations, checking and providing feedback are critical (Section V).

Second, runtime configurations are implemented differently than SPL configurations. Runtime configurations are mostly in the form of configuration parameters that load values from files or command lines; SPL configurations are typically in the form of preprocessors that determine modules to be included in the released binary.

Third, runtime configurations of cloud software are changed frequently (hundreds to thousands of times a day [1], [9],

[112]); SPL configurations are typically changed with product release cycles. This higher velocity of runtime configuration changes increases misconfiguration occurrences and makes checking, error handling, and logging critical.

Fourth, runtime configurations depend on the deployment environment, including machine resources (e.g., CPU, memory, and storage), operating systems (e.g., files, IP addresses, and ports), and workloads (data size and requests per seconds). In contrast, SPL configurations are often determined before software release or system deployment.

Lastly, runtime configurations have more complex data types (e.g., string and numeric) with different error patterns; SPL configurations are mostly boolean or enumerative types.

Certainly, ideas in SPL and variability modeling can be extended and applied to runtime configuration. We have discussed them in context of our analysis throughout the paper.

Appendix B: Replication Package

We release our research artifacts of this paper at:

https://github.com/xlab-uiuc/open-cevo

The artifacts include:

- the script code for collecting commits and JIRA issues;
- the annotated dataset of each categories (interface, usage, and document);
- documents for running the code and navigating the data.

We believe our study can be reproduced by different teams based on the taxonomy described in Figure 1 and Table II, together with the methodology described in Section III.

Author experience. All the authors have been working on software configuration research for several years, ranging from two to nine years. The authors have a good understanding of the four systems under study (HDFS, HBase, Spark, and Cassandra)—they have used those studied systems as subjects of evaluation in their prior research.

We expect that a similar level of experiences and expertises are needed for a team to reproduce the analysis (mainly categorization), including the understanding of the designs of the systems under study and the implementations in Java/Scala programming languages to understand the evolution. We believe a fair understanding of software configuration design and implementation is required, too.

Heuristics for commit identification and analysis. The heuristics are described in Section III-B2. The script code that implements the heuristics can be found at:

https://github.com/xlab-uiuc/open-cevo/tree/main/code

We documented how to run the code and the basic regular expressions used at the `README.md` file.

Identification/categorization of the rationales for parameterization. We identify and categorize the rationales for parameterization based on the descriptions (from the reporter) and the discussions (among the developers) documented in the JIRA issues. We developed our categories bottom-up instead of imposing a taxonomy *ex ante*. Specifically, we list all the rationales/reasons and then summarizing them into categories. We do not claim completeness of the categories.

We present an example from Spark, in which the commit (link) changed a constant value `100` to a configuration parameter, `spark.sql.codegen.cache.maxEntries`; the commit is associated with the JIRA issue, SPARK-24727. Quoting the JIRA issue description,

> "*The cache 100 in CodeGenerator is too small for realtime streaming calculation, although is ok for offline calculation. Because realtime streaming calculation is mostly more complex in one driver, and performance sensitive. I suggest spark support config for user with default 100, such as spark.codegen.cache=1000.*"

Based on the description, we conclude that the rationale behind the parameterization is that the constant value cannot meet the performance requirement of real-time application. So, it is categorized as "performance" in Table VI.

Identification on how developers find constants to parameterize. It is straightforward to identify the constants that were parameterized in a selected commit. The following commit (link) illustrates this point; the commit comes from SPARK-20950. We can see that the constant (`1024*1024`) was parameterized—the value was replaced by a program variable `diskWriteBufferSize` read from a configuration parameter.

```
- int DISK_WRITE_BUFFER_SIZE = 1024 * 1024
+ int diskWriteBufferSize =
+   conf.get(package$.MODULE$.
+     SHUFFLE_DISK_WRITE_BUFFER_SIZE)
- byte[] writeBuffer =
-     new byte[DISK_WRITE_BUFFER_SIZE]
+ byte[] writeBuffer =
+     new byte[diskWriteBufferSize]
```

For each commit that parameterized constants, we identify the intent of the parameterization, i.e., "how developers find constants to parameterize." Similar to the identification/categorization of the rationales for parameterization, we identify and categorize the reasons for how developers find constants to parameterize based on the descriptions (from the reporter) and the discussions (among the developers) documented in the JIRA issues. The process is identical—we list all the rationales/reasons and then summarizing them into categories. We do not claim completeness of the categories.

We present an example from Spark, in which the commit (link) parameterized the thread number of broadcast-exchange thread pool from a constant `128`. Based on the associated JIRA issue, SPARK-26601:

> "*Currently, thread number of broadcast-exchange thread pool is fixed and keepAliveSeconds is also fixed as 60s. But some times, if the thread object do not GC quickly it may caused server(driver) OOM. In such case, we need to make this thread pool configurable.*"

Therefore, we conclude that the parameterization is to avoid OOMs, a type of "failures" (Section IV-A1).

Identification/categorization of the reasons for changing default values. Similar to the identification/categorization of the rationales for parameterization, we identify and categorize the reasons for changing default values based on the descriptions (from the reporter) and the discussions (among the developers) documented in the JIRA issues. The process is identical—we list all the rationales/reasons and then summarizing them into categories. We do not claim completeness of the categories.

We present an example from HDFS. The commit (link) changed the default value of `dfs.namenode.edits.asynclogging` from `off` to `on`. The commit is associated with the JIRA issue, HDFS-12603, which in turn was linked to HDFS-7964. Quoting the discussion:

> "*It was off by default due to concerns about correctness. We have been running it in production for quite a while with no issues so far.*"

We conclude that the commit enabled a previously-disabled feature flag (Section IV-B).

Definitions of four levels of messages and feedback quality. The definitions are developed based on how the messages are changed, i.e., how developers improved the original messages. Specifically, we summarized what additional information is added to the original message, which can be categorized into the four levels. The following is a commit (link) that changed the message which improved the message quality from L3 to L4 defined in Table X.

```
-  resp.getWriter().write(
-  "ASYNC_PROFILER_HOME is not set.");
+  resp.getWriter().write(
+  "ASYNC_PROFILER_HOME is not set." +
+    "Please ensure prerequsites for the Profiler" +
+    "Servlet have been installed and the" +
+    "environment is properly configured." +
+    "For more information please see" +
+    "http://hbase.apache.org/book.html#profiler");
```

Handling commits that change multiple categories. One commit could revise multiple categories in our taxonomy (Figure 1 and Table II). In total, there are 26 commits that revise multiple categories. For those commits, we study them in each category independently. For example, the following commit (link) changes the default value of `spark.master.rest.enabled`; meanwhile, it also adds the checking code for `SPARK_AUTH_SECRET_CONF`.

```
-  val restServerEnabled = conf.getBoolean(
-    "spark.master.rest.enabled", true)
+  val restServerEnabled = conf.getBoolean(
+    "spark.master.rest.enabled", false)
...

+  authKey = SecurityManager.SPARK_AUTH_SECRET_CONF
+    require(conf.getOption(authKey).isEmpty
+     || !restServerEnabled,
+       s"The RestSubmissionServer does" +
+        "not support authentication.")
```

Therefore, we study this commit in both $\text{Mod}_{\text{DefaultValue}}$ (Section IV-B) and Check (Section V-A).

ACKNOWLEDGEMENT

We thank Xiangbing Huang, Xudong Sun, Sam Cheng, Jack Chen, and Darko Marinov for discussions. The research was mainly conducted when Zhang was a visiting student at UIUC, supported by China Scholarship Council. Zhang, He, Li, and Dong were supported in part of National Key R&D Program of China No. 2017YFB1001802; NSFC No. 61872373 and 61872375. Xu was supported in part of NSF 1816615.

REFERENCES

[1] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," in *SOSP*, 2015.

[2] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. 2018.

[3] J. Shieber, "Facebook blames a server configuration change for yesterday's outage." https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/, 2019.

[4] G. Amvrosiadis and M. Bhadkamkar, "Getting Back Up: Understanding How Enterprise Data Backups Fail," in *USENIX ATC*, 2016.

[5] 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 *SOSP*, 2011.

[6] S. Kendrick, "What takes us down?," *USENIX ;login:*, vol. 37, no. 5, 2012.

[7] A. Rabkin and R. Katz, "How Hadoop Clusters Break," *IEEE Software*, vol. 30, no. 4, 2013.

[8] 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 *SoCC*, 2016.

[9] B. Maurer, "Fail at Scale: Reliability in the Face of Rapid Change," *CACM*, vol. 58, no. 11, 2015.

[10] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?," in *USITS*, 2003.

[11] K. Nagaraja, F. Oliveira, R. Bianchini, R. Martin, and T. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *OSDI*, 2004.

[12] 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 *FSE*, 2015.

[13] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *SOSP*, 2013.

[14] T. Xu, H. M. Naing, L. Lu, and Y. Zhou, "How Do System Administrators Resolve Access-Denied Issues in the Real World?," in *CHI*, 2017.

[15] T. Xu, V. Pandey, and S. Klemmer, "An HCI View of Configuration Problems," *arXiv:1601.01747*, 2016.

[16] 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 *OSDI*, 2016.

[17] M. Attariyan and J. Flinn, "Automating Configuration Troubleshooting with Dynamic Information Flow Analysis," in *OSDI*, 2010.

[18] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, 2012.

[19] M. Attariyan and J. Flinn, "Using Causality to Diagnose Configuration Bugs," in *USENIX ATC*, 2008.

[20] S. Zhang and M. D. Ernst, "Automated Diagnosis of Software Configuration Errors," in *ICSE*, 2013.

[21] 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 *ASPLOS*, 2014.

[22] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic Misconfiguration Troubleshooting with PeerPressure," in *OSDI*, 2004.

[23] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," in *LISA*, 2003.

[24] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic Automated Language Learning for Configuration Files," in *CAV*, 2016.

[25] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," in *OOPSLA*, 2017.

[26] Z. Dong, A. Andrzejak, and K. Shao, "Practical and Accurate Pinpointing of Configuration Errors using Static Analysis," in *ICSME*, 2015.

[27] P. Huang, W. Bolosky, A. Sigh, and Y. Zhou, "ConfValley: A systematic configuration validation framework for cloud services," in *EuroSys*, 2015.

[28] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud," in *Middleware*, 2017.

[29] M. Sayagh, N. Kerzazi, and B. Adams, "On Cross-stack Configuration Errors," in *ICSE*, 2017.

[30] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *OSDI*, 2020.

[31] D. Norman, "Design Rules Based on Analyses of Human Error," *CACM*, vol. 26, no. 4, 1983.

[32] D. Norman, "Design principles for human-computer interfaces," in *CHI*, 1983.

[33] R. A. Maxion and R. W. Reeder, "Improving User-Interface Dependability through Mitigation of Human Error," *JHCS*, vol. 63, no. 1-2, 2005.

[34] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software Configuration Engineering in Practice Interviews, Survey, and Systematic Literature Review," *TSE*, vol. 46, no. 6, 2018.

[35] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, "Exploring Differences and Commonalities between Feature Flags and Configuration Options," in *ICSE SEIP*, 2020.

[36] S. Zhang and M. D. Ernst, "Which Configuration Option Should I Change?," in *ICSE*, 2014.

[37] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *FSE*, 2014.

[38] J. Bernardo, D. da Costa, and U. Kulesza, "Studying the impact of adopting continuous integration on the delivery time of pull requests," in *MSR*, 2018.

[39] M. Rigger, S. Marr, B. Adams, and H. Mössenböck, "Understanding GCC Builtins to Develop Better Tools," in *FSE*, 2019.

[40] L. P. Hattori and M. Lanza, "On the nature of commits," in *ASE*, 2008.

[41] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing Probabilistic Programming Systems," in *FSE*, 2018.

[42] A. Rabkin and R. Katz, "Static Extraction of Program Configuration Options," in *ICSE*, 2011.

[43] A. Rabkin and R. Katz, "Precomputing Possible Configuration Error Diagnosis," in *ASE*, 2011.

[44] M. Lillack, C. Kästner, and E. Bodden, "Tracking Load-time Configuration Options," in *ASE*, 2014.

[45] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Surveys*, vol. 47, no. 4, 2015.

[46] F. Behrang, M. B. Cohen, and A. Orso, "Users Beware: Preference Inconsistencies Ahead," in *FSE*, 2015.

[47] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the Linux Kernel Variability Model," in *SPLC*, 2010.

[48] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The Variability Model of the Linux Kernel," in *VaMoS*, 2010.

[49] D. Nesic, J. Krüger, S. Stanciulescu, and T. Berger, "Principles of Feature Modeling," in *ESEC/FSE*, 2019.

[50] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A Study of Variability Models and Languages in the Systems Software Domain," *TSE*, vol. 39, no. 12, 2013.

[51] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-based Software Product Lines," in *ICSE*, 2010.

[52] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *SIGMOD*, 2017.

[53] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and Auto-Adjusting Performance-Sensitive Configurations," in *ASPLOS*, 2018.

[54] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic Configuration of Internet Services," in *EuroSys*, 2007.

[55] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," in *ASPLOS*, 2018.

[56] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using Bad Learners to Find Good Configurations," in *FSE*, 2017.

[57] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-Influence Models for Highly Configurable Systems," in *FSE*, 2015.

[58] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *TSE*, vol. 46, no. 7, 2018.

[59] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh, "Scout: An Experienced Guide to Find the Best Cloud Configuration," *arXiv:1803.01296*, 2018.

[60] S. Duan, V. Thummala, and S. Babu, "Tuning Database Configuration Parameters with iTuned," in *VLDB*, 2009.

[61] Y. Zhu, J. Liu, M. Guo, Y. Bao, K. Song, and Z. Liu, "BestConfig: Tapping the Performance Potential of Systems via Configuration Adjustment," in *SoCC*, 2017.

[62] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal, "Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis," in *ASE*, 2017.

[63] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems," in *FSE*, 2018.

[64] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *WWW*, 2004.

[65] T. Ye and S. Kalyanaraman, "A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration," in *SIGMETRICS*, 2003.

[66] H. Herodotou, F. Dong, and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *SoCC*, 2011.

[67] T. Osogami and T. Itoko, "Finding Probably Better System Configurations Quickly," in *SIGMETRICS*, 2006.

[68] R. Krishna, V. Nair, P. Jamshidi, and T. Menzies, "Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE," *arXiv:1911.01817*, 2019.

[69] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-Aware Computing," in *ASPLOS*, 2011.

[70] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Towards Automatic Optimization of MapReduce Programs," in *CIDR*, 2011.

[71] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability Modeling in the Real: A Perspective from the Operating Systems Domain," in *ASE*, 2010.

[72] R. Barrett, E. Kandogan, P. P. Maglio, E. Haber, L. A. Takayama, and M. Prabaker, "Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices," in *CSCW*, 2004.

[73] L. Takayama and E. Kandogan, "Trust as an Underlying Factor of System Administrator Interface Choice," in *CHI*, 2006.

[74] E. M. Haber and J. Bailey, "Design Guidelines for System Administration Tools Developed through Ethnographic Field Study," in *CHI*, 2007.

[75] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "PrefFinder: Getting the Right Preference in Configurable Software Systems," in *ASE*, 2014.

[76] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *OSDI*, 2014.

[77] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *FSE*, 2020.

[78] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations," in *USENIX ATC*, 2020.

[79] C. Li, S. Wang, H. Hoffmann, and S. Lu, "Statically Inferring Performance Properties of Software Configurations," in *EuroSys*, 2020.

[80] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where Do Configuration Constraints Stem from? An Extraction Approach and An Empirical Study," *TSE*, vol. 99, 2015.

[81] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining Configuration Constraints: Static Analyses and Empirical Results," in *ICSE*, 2014.

[82] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Tech. Rep. CMU/SEI-90-TR-021, SEI, CMU, 1990.

[83] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki, "Reverse Engineering Feature Models," in *ICSE*, 2011.

[84] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating Range Fixes for Software Configuration," in *ICSE*, 2012.

[85] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range Fixes: Interactive Error Resolution for Software Configuration," *TSE*, vol. 41, no. 6, 2015.

[86] 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 *OSDI*, 2012.

[87] D. Yuan, S. Park, and Y. Zhou, "Characterising Logging Practices in Open-Source Software," in *ICSE*, 2012.

[88] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," in *ASPLOS*, 2011.

[89] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error diagnosis by connecting clues from run-time logs," in *ASPLOS*, 2010.

[90] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis," in *USENIX ATC*, 2015.

[91] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where Do Developers Log? An Empirical Study on Logging Practices in Industry," in *ICSE*, 2015.

[92] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *ICSE*, 2013.

[93] C. Sun, V. Le, and Z. Su, "Finding and Analyzing Compiler Warning Defects," in *ICSE*, 2016.

[94] T. Barik, J. Witschey, B. Johnson, and E. R. Murphy-Hill, "Compiler Error Notifications Revisited: An Interaction-first Approach for Helping Developers More Effectively Comprehend and Resolve Error Notifications," in *ICSE*, 2014.

[95] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code," in *SOSP*, 2001.

[96] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *USENIX Security*, 2008.

[97] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or Bad Comments? */," in *SOSP*, 2007.

[98] S. H. Tan, D. Marinov, L. Tan, and G. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *ICST*, 2012.

[99] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. Gall, "Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation," *TSE*, 2018.

[100] H. Zhong and Z. Su, "Detecting API Documentation Errors," in *OOPSLA*, 2013.

[101] E. Wong, J. Yang, and L. Tan, "AutoComment: Mining Question and Answer Sites for Automatic Comment Generation," in *ASE*, 2013.

[102] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis," in *ICSE*, 2020.

[103] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *ASE*, 2010.

[104] S. Zhang and M. D. Ernst, "Proactive Detection of Inadequate Diagnostic Messages for Software Configuration Errors," in *ISSTA*, 2015.

[105] N. Meng, S. Nagy, D. D. Yao, W. Zhuang, and G. A. Argoty, "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *ICSE*, 2018.

[106] C. Xiang, Y. Wu, B. Shen, M. Shen, H. Huang, T. Xu, Y. Zhou, C. Moore, X. Jin, and T. Sheng, "Towards Continuous Access Control Validation and Forensics," in *CCS*, 2019.

[107] T. Xu and D. Marinov, "Mining Container Image Repositories for Software Configurations and Beyond," in *ICSE NIER*, 2018.

[108] M. Sayagh, Z. Dong, A. Andrzejak, and B. Adams, "Does the Choice of Configuration Framework Matter for Developers? Empirical Study on 11 Java Configuration Frameworks," in *SCAM*, 2017.

[109] M. Raab and G. Barany, "Challenges in Validating FLOSS Configuration," in *OSS*, 2017.

[110] E. Horton and C. Parnin, "V2: Fast Detection of Configuration Drift in Python," in *ASE*, 2019.

[111] C. Wen, Y. Zhang, X. He, and N. Meng, "Inferring and applying def-use like configuration couplings in deployment descriptors," in *ASE*, 2020.

[112] S. Mehta, R. Bhagwan, R. Kumar, B. Ashok, C. Bansal, C. Maddila, C. Bird, S. Asthana, and A. Kumar, "Rex: Preventing bugs and misconfiguration in large services using correlated change analysis," in *NSDI*, 2020.

[113] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the Linux kernel," *TSE*, 2018.

[114] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 1, 2006.

[115] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *ICSE*, 2014.

[116] S. M. Fouché, M. B. Cohen, and A. Porter, "Incremental Covering Array Failure Characterization in Large Configuration Spaces," in *ISSTA*, 2009.

[117] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *ISSTA*, 2008.

[118] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems," in *ICSE*, 2010.

[119] C. Song, A. Porter, and J. S. Foster, "iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees," in *ICSE*, 2012.