Test-Case Prioritization for Configuration Testing

Runxiang Cheng
University of Illinois
Urbana-Champaign, IL, USA
rcheng12@illinois.edu

Lingming Zhang
University of Illinois
Urbana-Champaign, IL, USA
lingming@illinois.edu

Darko Marinov
University of Illinois
Urbana-Champaign, IL, USA
marinov@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

ABSTRACT

Configuration changes are among the dominant causes of failures of large-scale software system deployment. Given the velocity of configuration changes, typically at the scale of hundreds to thousands of times daily in modern cloud systems, checking these configuration changes is critical to prevent failures due to misconfigurations. Recent work has proposed configuration testing, *Ctest*, a technique that tests configuration changes together with the code that uses the changed configurations. Ctest can automatically generate a large number of ctests that can effectively detect misconfigurations, including those that are hard to detect by traditional techniques. However, running ctests can take a long time to detect misconfigurations. Inspired by traditional test-case prioritization (TCP) that aims to reorder test executions to speed up detection of regression code faults, we propose to apply TCP to reorder ctests to speed up detection of misconfigurations. We extensively evaluate a total of 84 traditional and novel ctest-specific TCP techniques. The experimental results on five widely used cloud projects demonstrate that TCP can substantially speed up misconfiguration detection. Our study provides guidelines for applying TCP to configuration testing in practice.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Software reliability**.

KEYWORDS

Test prioritization, configuration, software testing, reliability

ACM Reference Format:
Runxiang Cheng, Lingming Zhang, Darko Marinov, and Tianyin Xu. 2021. Test-Case Prioritization for Configuration Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464810

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464810

1 INTRODUCTION

Besides source-code changes, configuration changes are among the dominant causes of failures in large-scale software system deployments. In fact, configuration changes can be much more frequent than code changes. Many companies are deploying configuration changes to production systems hundreds to thousands of times a day [31, 33, 55, 64], hence *misconfigurations* become inevitable. For example, 16% of the service-level incidents at Facebook are induced by configuration changes [60], including major outages that turn down the entire service [21, 56], and misconfigurations were reported as the second largest cause of service disruptions in a main Google service [3]. The prevalence and severity of misconfigurations have been repeatedly reported by many failure studies [14, 23, 35, 37, 46, 68, 69, 73, 74].

Recently, Ctest has been proposed as a promising technique for configuration testing, i.e., testing a configuration before deployment [59, 70]. Ctest can effectively detect misconfigurations. The key idea of configuration testing is to connect configuration changes to software tests, so that configuration changes can be tested *in the context* of code affected by the changes. In this way, configuration testing can reason about the program behavior under the actual configuration values to be deployed and detect sophisticated misconfigurations that can hardly be detected by rule-based validation [4, 7, 17, 42, 60] or data-driven approaches [33, 38, 53, 54, 62, 66, 67, 76, 78]. Attractively, our prior work [59] shows that configuration test cases, or *ctests*, can be generated by parameterizing existing software tests abundant in mature software projects—up to 83.2% of existing tests can be transformed into ctests.

At a high level, a ctest is a software test parameterized by a set of configuration parameters. Running a ctest instantiates each parameter with a concrete value (e.g., the default value, the current value in production, or a new value to be deployed to production). Given a configuration change, all the ctests which are parameterized by at least one of the changed parameters are selected to run. Because one configuration parameter can parameterize many ctests, a configuration change can require running a large number of ctests. For example, some configuration changes from the HDFS project require running more than 2,000 ctests on average, which is over half of the total number of tests in that project [59]. Overall, in the Ctest dataset of five open-source projects (HCommon, HDFS, HBase, ZooKeeper, and Alluxio) [59], the number of ctests per configuration parameter is 1–3,069 (average 821), and a configuration change modifies 1–29 (average 6) parameters.

One main challenge in adopting configuration testing in continuous deployment is the time required to detect misconfigurations. This test-running time is on the critical path from the point where configuration changes are made to the point where they are deployed to production. For example, in the Ctest dataset, the time to run all ctests ranges from 20 minutes to 230 minutes (with an average of 97 *minutes*) per project. Given the velocity of configuration changes in modern deployment cycles [33, 60, 68], misconfigurations inevitably happen. With the large number of ctests to run before deployment, the time to detect the misconfiguration is crucial, because developers cannot start troubleshooting until the misconfiguration is detected. The time to detect the misconfiguration can greatly affect configuration deployment.

We are the first to address the cost of configuration testing using test-case prioritization (TCP). Traditionally, TCP aims to order regression tests to expose code bugs faster during software evolution. TCP has been extensively studied for over two decades [10, 48, 75]. For example, widely studied are the *total* TCP strategy [48] that favors tests covering more code elements and the *additional* TCP strategy [49] that favors tests covering more code elements not yet covered by already prioritized tests. Inspired by traditional TCP, we aim to leverage TCP techniques to order ctests to substantially speed up misconfiguration detection for configuration changes.

We extensively evaluate 84 TCP techniques on the large Ctest dataset [59], with 7,974 ctests for five open-source projects and 66 real-world configuration change files collected from public Docker images that have some misconfigured parameter values. Our experiments with configuration changes do *not* involve code changes, matching realistic scenarios where only a new configuration is about to be deployed. We start with 16 basic TCP techniques: (1) randomized as the baseline, (2) traditional techniques based on code coverage, (3) quickest-time-first (QTF) technique, (4) recently proposed techniques based on information retrieval (IR), and (5) our novel configuration-specific TCP techniques.

We next enhance the basic TCP techniques using two sources of inspiration. First, using the idea of *cost-cognizant* TCP [9, 30], we enhance basic TCP techniques with the test execution time to design *hybrid* TCP techniques. Second, inspired by cross-checking configurations of multiple system instances used in troubleshooting systems such as the Microsoft PSS [18, 63, 64], we design a new family of *peer-based* TCP techniques that consider the test outcomes of ctests on related configuration changes. The insight is to prioritize earlier ctests that detected misconfigurations of a parameter in peer deployments, because these ctests are likely effective for the parameter change regardless of the value. Following Microsoft PSS, our peer-based TCP techniques are privacy preserving and do not use potentially sensitive value information of peer deployments but use only parameter names.

Our study leads to the following key findings:

- Among basic techniques, QTF yields competitive performance and often outperforms sophisticated techniques (e.g., based on code coverage or IR) and even some configuration-specific techniques (e.g., based on parameter-coverage and stack traces) by up to 22% (using an APFDc-like metric, §4.2).
- Hybrid TCP techniques that enhance basic techniques with the test execution time improve the performance of basic techniques by up to 27%. Our results confirm that hybrid, *cost-cognizant* TCP techniques are effective, even in the new domain of configuration testing.
- Peer-based TCP techniques can outperform other techniques and improve the performance of TCP even further by 15%. The results encourage sharing configuration test outcomes for the same project: "*make friends and don't test alone!*"

Our paper makes the following contributions:

- Our work reduces the time to find misconfigurations, one of the main challenges of adopting configuration testing in real-world continuous deployment process.
- We evaluate 84 traditional and ctest-specific TCP techniques for configuration testing, and we have released our code and data at https://github.com/xlab-uiuc/ctest_prio_art.
- We analyze the effectiveness of TCP for ctests and find highly promising results for reducing the time to find test failures and thus detecting misconfigurations early.

2 BACKGROUND

Configuration testing is a testing technique for detecting misconfigurations (manifesting as failing tests) to prevent them from being deployed to production systems. The basic idea is to connect software tests with the specific configuration to be deployed. In this way, configuration testing can test configuration changes in the context of code that is affected by the changed configuration. A configuration test case (*ctest*) is parameterized by a set of configuration parameters. Running a ctest instantiates each of its input parameters with an actual configuration value to be deployed to production. Like regular software tests, ctests exercise the program and check (via assertions) that program behavior satisfies certain properties (e.g., correctness, performance, security). Figure 1 illustrates an example ctest from prior work [59].

Ctest (configuration testing) differs from approaches that explore *multiple* configurations, e.g., configuration-aware testing, combinatorial testing, or misconfiguration-injection testing [16, 22, 24, 32, 34, 44, 57, 72], which sample representative configurations or misconfigurations through systematic or random exploration of the enormous space of value combinations. Systematic exploration can be prohibitively expensive due to combinatorial explosion [34], while random exploration can have a low probability of covering all the values that will be deployed [32]. Ctest has neither the cost of systematic exploration nor the low coverage of random exploration. Ctest focuses on testing only *one* specific configuration that is to be deployed to the production system.

A ctest $\hat{t}(\hat{P})$ is parameterized by a set of configuration parameters $\hat{P}$. Running a ctest instantiates each parameter $p \in \hat{P}$ with a concrete value as an argument. $\hat{P}$ is typically a small subset of all the configuration parameters (denoted as $\mathbb{P}$).

A system configuration is defined as the values of *all* the configuration parameters, denoted as $C = \bigcup_{i=1..|\mathbb{P}|}\{(p_i \mapsto v_i)\}$, i.e., it assigns a value v_i to every parameter $p_i \in \mathbb{P}$. Running a ctest instantiates each parameter $p_i \in \hat{P}$ with its value in the system configuration v_i such that $(p_i \mapsto v_i) \in C$.

A configuration change updates the values of a subset of the configuration parameters. A configuration change is in the form of a configuration file diff D. To test a given D, not all available ctests are run. A ctest $\hat{t}(\hat{P})$ is selected to test a given D if at least

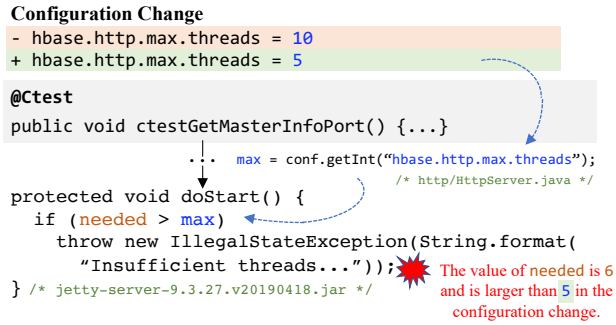

Figure 1: A ctest which exercises `doStart` with the value to be changed and detects the misconfiguration.

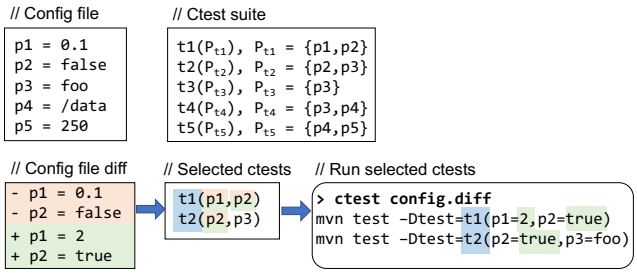

Figure 2: An overview of configuration testing for a configuration file diff. Only `t1` and `t2` are selected to run because they may be affected by the configuration change. The ctest framework [36] is built on top of Maven.

one configuration parameter in D is in the input parameter set $\hat{P}$. A configuration diff, D, passes if all *selected* ctests pass, and it fails if any selected ctest fails. Figure 2 gives an example of configuration testing for a given configuration file diff.

Overall, ctests check whether the configuration to be deployed has some misconfigurations, which will manifest as ctest failure(s). TCP for ctests pushes this further by trying to detect these misconfigurations, if any, as soon as possible by first running ctests that are more likely to fail for the new configuration.

3 TCP TECHNIQUES

We next present all the TCP techniques we study for reducing the cost of detecting misconfigurations in configuration testing. §3.1 presents basic TCP techniques that do not require peer configuration changes, while §3.2 presents basic TCP techniques that analyze the correlation between peer configuration changes and test failures to achieve more precise test prioritization. Lastly, §3.3 further introduces hybrid TCP techniques that combine basic peer-based or non-peer-based techniques with test execution time. Table 1 summarizes the notation for all evaluated TCP techniques.

3.1 Non-peer-based TCP

The non-peer-based TCP techniques include both traditional TCP techniques widely studied for regression testing (§3.1.1) and new TCP techniques we design for configuration testing (§3.1.2).

3.1.1 Traditional TCP techniques. We study the following traditional TCP techniques:

Code-Coverage-Based TCP. TCP techniques based on code coverage have been extensively evaluated [28, 48, 75] and are still widely used for comparisons against newly proposed techniques [40, 41]. Code-coverage-based TCP techniques determine the test execution order based on the code coverage of each test. For example, the *total* technique sorts tests in the descending order of the number of code elements (e.g., methods or statements) covered by each test, while the *additional* technique sorts tests in the descending order of the number of code elements covered by each test but uncovered by the already prioritized tests [49]. In the literature, code-coverage-based TCP has been widely studied at both the method and statement granularities [28]. Thus, we also evaluate *total* and *additional* code-coverage-based TCP at both method (denoted as CC^m_{tot} and CC^m_{add}) and statement granularity (denoted as CC^s_{tot} and CC^s_{add}).

IR-Based TCP. Techniques based on information retrieval (IR) have been recently proposed and shown effective in test-case prioritization [41, 51]. IR-based techniques transform the TCP problem into an IR problem and address it with off-the-shelf retrieval models (e.g., Tf-idf [52] and BM25 [47]). A typical IR-based technique extracts code tokens from test files to form a corpus of *documents*, and represents code change information (e.g., tokens extracted from code change diff) as the *query*. In this way, a similarity value can be computed between the query and each test document. Tests that are more similar to code changes are prioritized earlier to detect problematic changes faster. We implement and evaluate the IR_{high} and IR_{low} techniques with BM25, as it has shown the best results [41, 51].

QTF-Based TCP. The Quickest Time First (QTF) technique simply orders all the tests in the ascending order of their execution time in prior testing runs [50]. Although simple, the QTF technique has been shown to be competitive compared with state-of-the-art TCP techniques for regression testing [6]. Therefore, we also evaluate QTF in the context of configuration testing.

3.1.2 Configuration-specific TCP techniques. We further design the following TCP techniques specifically for configuration testing:

Parameter-Coverage-Based TCP. Inspired by traditional TCP techniques based on *code* coverage, we propose novel TCP techniques based on *parameter* coverage. Following the definition of ctest (§2), each ctest $\hat{t}(\hat{P})$ can test a non-empty set of input configuration parameters $\hat{P}$. We treat $\hat{P}$ as the parameters covered by $\hat{t}$. We propose *total* and *additional* TCP techniques based on such parameter coverage, denoted as PC_{tot} and PC_{add}, respectively.

We also consider the parameter change information to design *change-aware*, parameter-coverage-based TCP. For each configuration change D, the set of changed parameters is denoted P_D. For each ctest $\hat{t}(\hat{P})$, we determine its priority based on the set of covered changed parameters, i.e., $\hat{P} \cap P_D$. Change-aware parameter coverage prioritizes ctests that are more relevant to the configuration change, thus can potentially detect misconfigurations earlier. We evaluate both *total* and *additional* techniques based on change-aware parameter coverage, denoted as PC^D_{tot} and PC^D_{add}, respectively.

Stack-Trace-Based TCP. Different ctests may read and test the same parameter in different invocation contexts and thus may have different capabilities in detecting problematic parameter changes. For example, two ctests $\hat{t}_1(\hat{P}_1)$ and $\hat{t}_2(\hat{P}_2)$ may read $p \in \hat{P}_1 \cap \hat{P}_2$ in

Table 1: Notation for all evaluated TCP techniques

TCP Category	Notation
Traditional (§3.1.1)	
Method-level code-coverage-based	CC^m
Statement-level code-coverage-based	CC^s
IR-based with high tokenization	IR_{high}
IR-based with low tokenization	IR_{low}
Quickest time first	QTF
Configuration-specific (§3.1.2)	
Change-unaware parameter-coverage-based	PC
Change-aware parameter-coverage-based	PC^D
Change-unaware stack-trace-based	ST
Change-aware stack-trace-based	ST^D
Peer-based (§3.2)	
All configurations	$Conf^{all}$
Configurations sharing parameter changes	$Conf^{DP}$
Configurations sharing parameter coverage	$Conf^{PC}$
Configurations sharing root causes	$Conf^{RC}$
Shared parameter coverage with peers	$Para^{PC}$
Shared root causes with peers	$Para^{RC}$
Hybrid Models (§3.3)	
Divide-by-time hybrids	$*+T_{div}$
Break-tie-by-time hybrids	$*+T_{tie}$
Others	
Total techniques	$*_{tot}$
Additional techniques	$*_{add}$
Randomized order	Rand

different source code locations, and the invocation contexts can be used to prioritize the two reads. Thus, we use the invocation contexts for each parameter read for more precise ctest prioritization.

A ctest $\hat{t}(\hat{P})$ instantiates each parameter $p \in \hat{P}$ by reading its value from configuration file(s) via API calls provided by the configuration management class(es) in the system. The ctest infrastructure [36, 59] intercepts the configuration APIs and logs the stack trace of each API invocation during generation of ctests from regular tests (and not necessarily during ctest execution). The set of methods within the invocation contexts for all parameter reads of each test can be extracted from the stack traces and leveraged for TCP. We implement both the *total* and *additional* techniques based on such information, denoted as ST_{tot} and ST_{add}, respectively.

While ST_{tot} and ST_{add} consider all the methods from all stack traces where $\hat{t}$ reads all the parameters from $\hat{P}$, the change-aware variants for a configuration change D consider all the methods from all stack traces where $\hat{t}$ reads only the parameters from $\hat{P} \cap P_D$. The *total* and *additional* techniques for this change-aware variants are denoted as ST^D_{tot} and ST^D_{add}, respectively.

3.2 Peer-based TCP

We now present a family of new ctest TCP techniques, termed *peer-based TCP*, that consider the test outcomes of ctests from related, *peer* configurations. Data from peer systems have been used in troubleshooting systems such as the Microsoft PSS [18, 63, 64], e.g., *PeerPressure* utilizes configuration data from peer machines to infer root causes of misbehavior [64]. Inspired by this idea, peer-based TCP prioritizes ctests that detected misconfigurations of a parameter in peer deployments, as these ctests are likely to be effective for the parameter change regardless of the value.

Deploying peer-based TCP can be done via a server/database that receives, anonymizes, and stores failed configurations and ctest outcomes from internal or community sources, to be used for future prioritization, e.g., *PeerPressure* utilized the GeneBank database to troubleshoot misconfigurations at Microsoft [64]. Specifically, our peer-based TCP are privacy preserving and do not use potentially sensitive values of peer deployments.

The general definition of peer-based TCP is simple. Let D be a configuration change to be tested by a ctest suite T, and S be a set of peer configuration changes ($D \notin S$) that have been tested. A peer-based TCP technique orders T based on various statistics collected from S. Depending on the granularity of the peer analysis, we propose two categories of peer-based techniques, at the configuration granularity (§3.2.1) and the parameter granularity (§3.2.2). Given a ctest $\hat{t}$, D, and information from S, each technique computes a *set* of elements $X(\hat{t}, D, S)$ for the ctest; these sets can be ordered using a *total* (X_{tot}) or *additional* (X_{add}) approach, and we evaluate both on all categories of peer-based techniques.

We illustrate all our proposed techniques using the example shown in Figure 3. It contains a configuration change `D`, its ctest suite `T`, and a set of peer configuration changes `S`. Note that each change is with respect to some default configuration and lists parameters whose values changed. The root-cause information specifies the misconfigured parameter(s) that caused a ctest to fail on a configuration change (e.g., only `p3` caused `t1` to fail on `D1`). Empty cells indicate that the test passed. Thus, `T` has three types of orders for `D`: optimal (run passing `t3` last), sub-optimal (run `t3` second), and worst-case (run `t3` first).

3.2.1 Techniques at the configuration granularity. We now discuss TCP techniques based on peer configuration changes at different granularity levels:

All Configurations ($Conf^{all}$). The $Conf^{all}$ set of each ctest $\hat{t}(\hat{P})$ is simply the set of all peer configuration changes where $\hat{t}$ failed:

$$Conf^{all}(\hat{t}, D, S) = \{D' \in S \mid Fail(\hat{t}, D')\} \quad (1)$$

where $Fail(\hat{t}, D')$ indicates that $\hat{t}$ failed on a peer configuration change D'. For the example in Figure 3, $Conf^{all}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}, \texttt{D4}\}$, $Conf^{all}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$, and $Conf^{all}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}, \texttt{D4}, \texttt{D5}\}$. Thus, $Conf^{all}_{tot}$ orders `T` as `t3-t1-t2`, and $Conf^{all}_{add}$ can order `T` as `t3-t2-t1` or `t3-t1-t2`. According to the root causes of `D`, both techniques only produce worst-case orders of `T`.

$Conf^{all}$ is *change-unaware* and can prioritize earlier a ctest that failed many peer configuration changes even if they share no changed parameter(s) with D, degrading T's performance in detecting the misconfigurations in the parameters changed in D. Thus, all the following peer-based TCP techniques are *change-aware* and consider which parameters have changed for better prioritization.

Configurations Sharing Parameter Changes ($Conf^{DP}$). The $Conf^{DP}$ set of each ctest $\hat{t}(\hat{P})$ restricts the set to peer configuration

// Ctest suite

```
T = {t1, t2, t3}
t1(Pt1), Pt1 = {p2, p3, p4, p5}
t2(Pt2), Pt2 = {p1, p6}
t3(Pt3), Pt3 = {p2, p4}
```

// Peer config changes

```
S = {D1, D2, D3, D4, D5}
D1, PD1 = {p1, p2, p3, p4}
D2, PD2 = {p1, p2, p4}
D3, PD3 = {p1, p4}
D4, PD4 = {p4, p5, p6}
D5, PD5 = {p4}
```

// Current config change

```
D, PD = {p1, p2, p3}
```

// Root causes of ctest failures

	t1	t2	t3
D1	{p3}	{p1}	{p4}
D2	{p4}	{p1}	{p4}
D3	{p4}	{p1}	{p4}
D4	{p5}		{p4}
D5			{p4}
D	{p3}	{p1}	

Figure 3: An example to illustrate peer-based TCP

changes that have changed parameters in common[1] with D:

$$\text{Conf}^{DP}(\hat{t}, D, S) = \{D' \in S \mid P_{D'} \cap P_D \neq \{\} \wedge \text{Fail}(\hat{t}, D')\} \quad (2)$$

For our example, $\text{Conf}^{DP}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$, because $\texttt{P}_{\texttt{D1}} \cap \texttt{P}_\texttt{D} = \{\texttt{p1}, \texttt{p2}, \texttt{p3}\}$, $\texttt{P}_{\texttt{D2}} \cap \texttt{P}_\texttt{D} = \{\texttt{p1}, \texttt{p2}\}$, and $\texttt{P}_{\texttt{D3}} \cap \texttt{P}_\texttt{D} = \{\texttt{p1}\}$. Similarly, $\text{Conf}^{DP}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$ and $\text{Conf}^{DP}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$. Both Conf^{DP}_{tot} and Conf^{DP}_{add} can produce all 6 permutations of T because all 3 ctests have the same priority.

While more precise than Conf^{all}, Conf^{DP} could include D' when changed parameters in common between D' and D are not even read by $\hat{t}$. In this way, a larger set for Conf^{DP} may not indicate that $\hat{t}$ is more effective in detecting misconfigurations on the current changed parameters read by $\hat{t}$. Therefore, we next consider parameter coverage information for more precise TCP.

Configurations Sharing Parameter Coverage (Conf^{PC}). The Conf^{PC} set of each ctest $\hat{t}(\hat{P})$ further restricts the set to peer configuration changes that have changed parameters in common with D and also some parameter(s) in common read by $\hat{t}$:

$$\text{Conf}^{PC}(\hat{t}, D, S) = \{D' \in S \mid P_{D'} \cap P_D \cap \hat{P} \neq \{\} \wedge \text{Fail}(\hat{t}, D')\} \quad (3)$$

For our example, $\text{Conf}^{PC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}\}$ because $\texttt{P}_{\texttt{D1}} \cap \texttt{P}_\texttt{D} \cap \texttt{P}_{\texttt{t1}} = \{\texttt{p2}, \texttt{p3}\}$ and $\texttt{P}_{\texttt{D2}} \cap \texttt{P}_\texttt{D} \cap \texttt{P}_{\texttt{t1}} = \{\texttt{p2}\}$, while $\texttt{P}_{\texttt{D3}} \cap \texttt{P}_\texttt{D} \cap \texttt{P}_{\texttt{t1}} = \{\}$. Similarly, $\text{Conf}^{PC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$ and $\text{Conf}^{PC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}\}$. Both Conf^{PC}_{tot} or Conf^{PC}_{add} can order T as t2-t1-t3 or t2-t3-t1. Either technique has 50% probability of producing an optimal or sub-optimal order of T, and produces no worst-case order.

Conf^{PC} may still be imprecise when the exact parameter(s) that caused $\hat{t}$ to fail on D' are not in $P_{D'} \cap P_D \cap \hat{P}$, which happens when the root-cause parameter(s) of $\hat{t}$ on D' are not in P_D, thus not in $P_{D'} \cap P_D$. In such a scenario, even if the technique prioritizes ctests with larger Conf^{PC}, the misconfiguration detection efficiency on D may not improve simply because the root-cause parameter(s) of the peer configuration changes are not in P_D. Therefore, we next consider the root-cause parameter information.

Configurations Sharing Root Causes (Conf^{RC}). The Conf^{RC} set of each ctest $\hat{t}(\hat{P})$ further restricts the set to peer configuration changes whose root-cause misconfigured parameters are also changed in D:

$$\text{Conf}^{RC}(\hat{t}, D, S) = \{D' \in S \mid \text{RC}(\hat{t}, D') \cap P_D \neq \{\} \wedge \text{Fail}(\hat{t}, D')\} \quad (4)$$

$\text{RC}(\hat{t}, D')$ is the set of root-cause misconfigured parameter(s) that actually caused the failure of $\hat{t}$ in configuration change D'. Note that $\text{RC}(\hat{t}, D') \subseteq \hat{P}$ because a parameter must be read by $\hat{t}$ (i.e, in $\hat{P}$) to be a root cause of the failure of $\hat{t}$.

In Figure 3, $\text{Conf}^{RC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{D1}\}$ because only $\text{RC}(\texttt{t1}, \texttt{D1}) \cap \texttt{P}_\texttt{D} = \{\texttt{p3}\}$ is non-empty. Similarly, $\text{Conf}^{RC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{D1}, \texttt{D2}, \texttt{D3}\}$ and $\text{Conf}^{RC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\}$. Conf^{RC}_{tot} orders T as t2-t1-t3, and Conf^{RC}_{add} can order T as t2-t1-t3 or t2-t3-t1. The probability of producing an optimal order of T is 50–100%, and no worst-case order is produced.

While Conf^{RC} is more precise than the earlier peer-based techniques, it requires to maintain the root-cause information for all failed peer configuration changes. Developers could record such information while debugging misconfigurations, but such information may not always be available (§4.3).

3.2.2 Techniques at the parameter granularity. We now discuss our peer-based techniques based on individual parameters in peer configurations at different precision levels. Conf^{all} and Conf^{DP} techniques do not consider parameter coverage and thus have no parameter-granularity counterparts.

Shared Parameter Coverage with Peers (Para^{PC}). The Para^{PC} set of each ctest $\hat{t}(\hat{P})$ is the set of parameters from peer configuration changes in $\text{Conf}^{PC}(\hat{t}, D, S)$ described in §3.2.1:

$$\text{Para}^{PC}(\hat{t}, D, S) = \bigcup_{D' \in S,\ \text{Fail}(\hat{t}, D')} P_{D'} \cap P_D \cap \hat{P} \quad (5)$$

Collecting for each ctest the parameters instead of failed peer configuration changes explores another possibility where peer-based TCP could prioritize earlier ctests that failed on a relatively smaller number of peer configuration changes but a larger set of configuration parameters from the changes.

For our example, $\text{Para}^{PC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{p2}, \texttt{p3}\}$, $\text{Para}^{PC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{p1}\}$, and $\text{Para}^{PC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\texttt{p2}\}$. Para^{PC}_{tot} orders T as t1-t2-t3 or t1-t3-t2. Para^{PC}_{add} orders T as t1-t2-t3. The probability of producing optimal orders of T is 50–100%, with no worst-case order produced, which is an overall improvement to the counterpart (Conf^{PC}) from §3.2.1.

Shared Root Causes with Peers (Para^{RC}). The Para^{RC} set of each test $\hat{t}(\hat{P})$ is the set of parameters from peer configuration changes in Conf^{RC} $(\hat{t}, D, S)$ described in §3.2.1:

$$\text{Para}^{RC}(\hat{t}, D, S) = \bigcup_{D' \in S,\ \text{Fail}(\hat{t}, D')} \text{RC}(\hat{t}, D') \cap P_D \quad (6)$$

For our example, $\text{Para}^{RC}(\texttt{t1}, \texttt{D}, \texttt{S}) = \{\texttt{p3}\}$, $\text{Para}^{RC}(\texttt{t2}, \texttt{D}, \texttt{S}) = \{\texttt{p1}\}$, and $\text{Para}^{RC}(\texttt{t3}, \texttt{D}, \texttt{S}) = \{\}$. Both Para^{RC}_{tot} and Para^{RC}_{add} can order T as t1-t2-t3 or t2-t1-t3. The probability of producing optimal orders of T is thus 100%, which improves over the counterpart (Conf^{RC}) from §3.2.1.

3.3 Hybrid TCP

Various TCP techniques have been reported to benefit by additionally considering test execution time [9, 30, 41, 50]. For example, the *cost-cognizant additional* code-coverage-based technique [30], which considers the additional code coverage per time unit for each test, can substantially improve the *additional* technique in terms of the time for detecting regression faults. Therefore, besides all the basic TCP techniques introduced in §3.1–3.2, we introduce hybrid techniques that combine the basic techniques with test execution time. Inspired by the prior work in cost-cognizant TCP [30, 41, 50],

[1] Note that it considers only parameter *names* and *not values*.

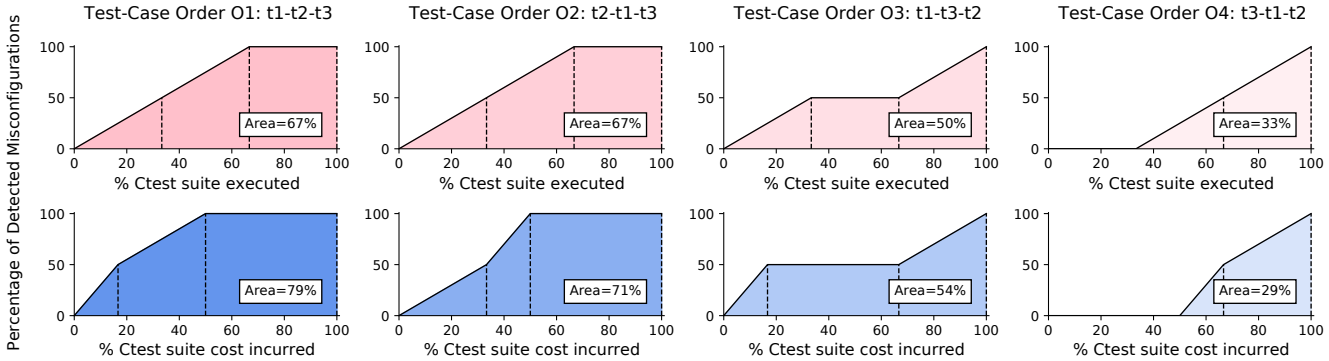

Figure 4: An example to illustrate APMD (first row) and APMDc (second row) for four test-case orders

we define and implement two generic cost-cognizant hybrid TCP models. We apply both models to all aforementioned TCP techniques to construct hybrid TCP techniques, and evaluate their prioritization effectiveness for ctests.

3.3.1 Divide-By-Time. Following the traditional cost-cognizant TCP techniques, the Divide-by-time (T_{div}) model constructs hybrid TCP techniques that prioritize tests in the descending order of the input tests' priority values per time unit, i.e., the original priority values divided by the test execution time. For example, a hybrid *additional* code-coverage TCP technique with T_{div} model (CC^{m}_{add}+T_{div}) prioritizes the test with the largest value of the number of uncovered methods divided by the test execution time.

3.3.2 Break-Tie-By-Time. We further study how to use time information in the Break-tie-by-time (T_{tie}) model. It constructs hybrid TCP techniques that order tests that are "tied" by the basic TCP technique (i.e., multiple tests have the same priority score) in the ascending order of their test execution time (as QTF). For example, hybrid technique CC^{m}_{tot}+T_{tie} orders the tied tests with QTF when multiple tests have the same amount of covered methods.

4 EXPERIMENTAL SETUP

4.1 Research Questions

In this study, we aim to answer the following research questions:

- **RQ1:** How do basic non-peer-based TCP techniques perform in detecting real-world misconfigurations?
- **RQ2:** How do hybrid non-peer-based TCP techniques perform compared with the basic non-peer-based techniques?
- **RQ3:** How do peer-based TCP techniques perform compared to non-peer-based TCP techniques?

4.2 Metrics

Common metrics to evaluate traditional TCP techniques are Average Percentage of Faults Detected (APFD) and Average Percentage of Faults Detected per Cost (APFDc) [75]. APFDc is a *cost-aware* variant of APFD that considers the cost of test executions [9, 30]. In the context of configuration testing, however, test failures are caused by misconfigurations and not (code) regression faults. Thus, we adapted the definition of APFD and APFDc to derive two new metrics for evaluating TCP techniques for configuration testing: Average Percentage of Misconfigurations Detected (APMD) and Average Percentage of Misconfigurations Detected per Cost (APMDc). The *only* difference in the definitions is that APFD and APFDc consider code bugs, while our metrics consider misconfigurations. Higher APMD and APMDc values (i.e., closer to 1.0) indicate all misconfigurations are detected earlier, while lower values (i.e., closer to 0.0) indicate all misconfigurations are detected later.

We illustrate APMD and APMDc for the following example scenario. Let T = {t1, t2, t3} be a ctest suite for a configuration change D, P_D = {p1, p2}; t1 failed on p1, t2 failed on p2, and t3 passed; the execution costs of t1, t2, and t3 are 1, 2, and 3 seconds, respectively. Figure 4 illustrates the APMD and APMDc values for four orders (i.e., O1, O2, O3, and O4) of T; from left to right, O1 and O2 are optimal (run passing t3 last), O3 is sub-optimal (runs t3 second), O4 is the worst-case (runs t3 first).

Average Percentage of Misconfigurations Detected (APMD). APMD is our adaption of APFD [48] in the context of configuration testing. Let n be the number of configuration tests to be run, m be the number of misconfigured parameters in the configuration change, and TF_i be the position (in the order) of the first failed configuration test that detects the i^{th} misconfigured parameter:

$$APMD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{7}$$

APMD computes the area under the curve between the percentage of detected misconfigurations in a configuration change and the percentage of the test suite executed, as illustrated in Figure 4. Note that a larger area always implies faster overall detection for *all* misconfigurations in the current configuration change. For example, O1 detects 50% of the misconfigurations in D (i.e., p1) after executing 33.3% of T (i.e., t1), and O1 detects 100% of the misconfigurations in D (i.e., p1, p2) after executing 66.7% of T (i.e., t1, t2). Thus, the APMD value of O1 is 67% as $1 - \frac{1+2}{3 \cdot 2} + \frac{1}{2 \cdot 3} = 0.67$ using Formula 7. However, like APFD, APMD is *cost-unaware*. Although O1 and O2 have the same APMD value, O1 is actually more cost-effective than O2 because O1 halves the cost to detect the first misconfiguration compared to O2.

Average Percentage of Misconfigurations Detected per Cost (APMDc). APMDc considers the cost, as in APFDc, which commonly uses test execution time [6, 11]. Let n, m, and TF_i be the same

as for APMD, and t_j be the execution time[2] of the j^{th} configuration test in the prioritized order:

$$APMDc = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j \times m} \quad (8)$$

Similar to APMD, APMDc computes the area under the curve between the percentage of detected misconfigurations in a configuration change and the percentage of its test suite *cost* incurred, as illustrated in Figure 4. For example, the total cost of T is 6 seconds; O1 detects 50% of the misconfigurations in D after incurring 17% of the total cost (i.e., 1 second from t1), and O1 detects 100% of the misconfigurations in D after incurring 50% of the total cost (i.e., 1 second from t1 and 2 seconds from t2). Thus, the APMDc value of O1 is 79% as $\frac{(1+2+3-\frac{1}{2}\cdot 1)+(2+3-\frac{1}{2}\cdot 2)}{(1+2+3)\cdot 2} = 0.79$ using Formula 8. The APMDc value of O2 is lower than that of O1, showing that APMDc can properly distinguish the more cost-effective order.

APMDc, like APFDc, more precisely captures the cost/time that developers would actually experience to detect *all* misconfigurations. Prior studies [6, 30] show that APFD can rank TCP techniques for regression faults differently than APFDc, and thus APFD is less reliable. We still evaluate both APMD and APMDc to check if the same holds for TCP techniques in the new application domain of configuration testing.

4.3 Dataset Collection

We build our evaluation dataset from the Ctest dataset [59], which contains 66 configuration changes with misconfigurations collected from real-world Docker images on Docker Hub [8, 71] for five widely-used projects: HCommon, HDFS, HBase, ZooKeeper, and Alluxio. The dataset also includes ctests for these projects. To compute APMD and APMDc, we ran ctests on all configuration changes and collected test outcomes and execution time.

We also identified the root-cause misconfigured parameter(s) for each test failure. Root-cause information is necessary to precisely compute APMD and APMDc for any TCP technique. (Prior research on regression testing has likewise had to map each test failure to the code fault(s) to compute APFD and APFDc [9, 30].) It is also necessary for constructing peer information for some peer-based TCP techniques (§3.2). Automated root-cause localization such as delta debugging [77] is *not* applicable because misconfigurations are not monotone due to configuration dependencies [7]. While several advanced misconfiguration-diagnosis techniques exist [1, 2, 45, 65, 82, 83], we manually localized the root causes to ensure the precision; most failure-inducing misconfigured parameters can be easily identified as root causes by inspecting failure logs. Besides the techniques that need root causes, all others are fully *automatic*. We excluded flaky tests from the dataset using best-effort reruns [5]. Table 2 shows the version, number of configuration changes, and average numbers of parameters, misconfigured parameters, and ctests per change of each project.

[2]Note that the time for APMDc is measured when running tests on the changed configuration, while the time used to prioritize tests (in QTF and hybrid techniques) is from running tests prior to the change.

Table 2: Configuration Change Dataset

Project	Ver.	#Changes	Avg #Params		Avg #Ctests
			All	Misconf	
HCommon	2.8.5	20	3.75	1.05	955.75
HDFS	2.8.5	16	5.19	1.31	1680.12
HBase	2.2.2	12	8.33	1.92	1254.25
ZooKeeper	3.5.6	14	6.57	1.71	74.36
Alluxio	2.1.0	4	13.75	1.25	949.00

4.4 Implementation

We implemented the main logic of all the studied TCP techniques in Python 3. Our infrastructure for test information collection and test prioritization is written in Java and Python.

4.4.1 Test Information Collection. We next discuss how we collected the necessary test information required by the studied TCP techniques. We used OpenClover [39] to collect code coverage at statement and method granularity (§3.1.1). To collect ctest execution time (§3.1.1, §3.3), we ran each ctest 5 times *prior* to configuration changes on the same machine, and used the averages as the time for prioritization. Execution times reported as 0.000 by Maven are changed to 0.001 because Maven rounds off time to 3 decimal places. For IR data (§3.1.1), we implemented a parser in Java 8 with JavaParser 3.18.0 [19] to collect tokens from test class files for all evaluated projects. We also performed an automated step of ctest generation with the open-sourced Ctest prototype [36] to collect invocation contexts for stack-trace-based TCP techniques (§3.1.2). We directly collected parameter coverage (§3.1.2) from open-sourced ctests [36]. Inspired by cross validation [58], for each configuration change in the dataset, we treated the other configuration changes from the same project as its peer configuration changes (§3.2).

4.4.2 Test Prioritization. Because most of the studied TCP techniques are built based on the traditional *total* and *additional* techniques, we implemented generic *total* and *additional* TCP functions following the traditional definitions. We also implemented the QTF TCP technique according to the traditional definition.

For IR-based techniques (§3.1.1), the choice of retrieval model and the approach to construct data objects can substantially affect the performance [41, 51]. Our IR-based TCP techniques used the BM25 retrieval model [47], as well as $High_{token}$ and Low_{token} for data-object construction, which have been demonstrated to achieve state-of-the-art performance by Peng *et al.* [41]. Specifically, our IR_{high} TCP technique used the $High_{token}$ construction, where a document only contains identifiers from a test file. Similarly, our IR_{low} TCP technique used the Low_{token} construction, where a document contains identifiers, comments, and string literals from a test file. We collected documents at test-case level utilizing Saha *et al.*'s approach [51], treating each test method as a test case, as common in JUnit. We processed documents following standard tokenization steps [41]. Unlike code changes, which can contain a variety of elements, a configuration change only contains names and values of the changed parameters. To construct query for each configuration change, we only use tokenized names of the changed parameters, because actual configuration values are often too specific to be found in the test code.

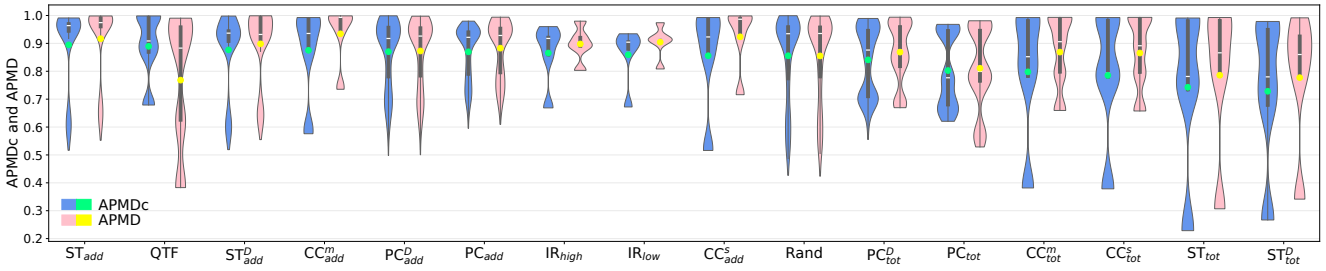

Figure 5: Distribution of APMDc and APMD values for basic non-peer-based TCP techniques (sorted by average APMDc)

4.5 Experimental Procedure

To compare all the studied TCP techniques, we also implemented a randomized TCP technique (denoted as Rand) to serve as baseline, which shuffles ctests with a random seed. For all studied TCP techniques with no break-tie strategy specified, ties are also broken with random seeds. Thus, to amount for different results from randomization, we ran each TCP technique on every configuration change 100 times, each time with a different seed. Specifically, for each TCP technique, we did the following: (1) load the collected configuration change dataset, i.e., ctest outcome, execution time, root-cause analysis results under configuration changes, etc. (§4.3); (2) load the test information (§4.4) for the current technique; (3) select a configuration change D that has not been run under the current technique; (4) initialize a random seed; (5) apply current technique to order the ctest suite of D; (6) compute APMD and APMDc of the ctest order based on the collected ctest outcome, execution time, and root causes; (7) repeat steps (4)–(6) 100 times; (8) repeat steps (3)–(7) on all 66 configuration changes.

In total, we evaluated 84 TCP techniques for configuration testing: 16 basic non-peer-based techniques, of which 15 are described in §3.1 and 1 is randomized baseline; 12 basic peer-based techniques described in §3.2; 32 hybrid non-peer-based techniques, of which 16 each use T_{div} and T_{tie} models (§3.3); and 24 hybrid peer-based ones. In total, we performed 554,400 (84*66*100) unique TCP executions.

5 RESULTS AND ANALYSIS

5.1 RQ1: Basic Non-Peer-Based TCP

This RQ compares non-peer-based traditional and configuration-specific TCP techniques on APMD and APMDc. In Figure 5, each violin plot and its embedded box plot show the distribution of APMD or APMDc values per project per run for each TCP technique. Each volin/box plot represents 500 (5*100) data points, for five projects and 100 random seeds. The white bar in each box plot shows the median, while the dot shows the (arithmetic) mean over all the data points for each TCP technique.

We further show the Tukey HSD test [61] results in Table 3. Tukey HSD is a post-hoc test based on the studentized range distribution; it compares all possible pairs of means to find out which specific groups' means (compared with each other) are significantly different. We performed this test on APMD and APMDc values to check for statistically significant differences among the studied TCP techniques [41]. In the table, Column "Average" shows the mean APMDc ("A.c") and APMD ("A.") values per technique (same as the dots in Figure 5). Importantly, Column "Group" presents the results of the Tukey HSD test. Tukey HSD puts techniques into different groups if they have statistically significant differences. Groups are named by capital letters, where "A" denotes the best group, and the performance degrades in alphabetical order. A technique having multiple letters has performance between these letter groups. From the results, we make the following observations.

Table 3: Results for basic non-peer-based TCP

TCP	Average		Group	
	A.c	A.	A.c	A.
ST_{add}	.895	.917	A	AB
QTF	.890	.768	AB	G
ST^{D}_{add}	.877	.898	ABC	BCD
CC^{m}_{add}	.875	.934	ABC	A
PC_{add}	.870	.883	ABC	CDE
PC^{D}_{add}	.870	.873	ABC	CDE
IR_{high}	.865	.898	ABC	BCD
IR_{low}	.859	.904	ABC	ABC
CC^{s}_{add}	.856	.924	BC	AB
Rand	.856	.855	BC	E
PC^{D}_{tot}	.841	.869	C	DE
PC_{tot}	.803	.811	D	F
CC^{m}_{tot}	.798	.869	D	DE
CC^{s}_{tot}	.785	.865	D	E
ST_{tot}	.743	.786	E	FG
ST^{D}_{tot}	.728	.777	E	G

5.1.1 Total vs. Additional. We can observe that *additional* techniques tend to outperform *total* ones on APMD and APMDc. For example, stack-trace-based TCP has the highest average APMDc value (0.895) among all studied techniques when using the *additional* strategy, but it has one of the lowest average APMDc values (0.743) when using the *total* strategy. Similar findings can be observed for code-coverage-based TCP on the APMD values, as well as other studied techniques. The Tukey HSD test results also confirm our observation, e.g., for APMDc, almost all *additional* techniques are in better Tukey HSD groups than Rand, while all *total* techniques are in worse groups. The key reason is that the *additional* strategy considers the impact of already prioritized tests and tends to execute more diverse tests, which can expose misconfigurations earlier. This finding is consistent with prior studies on traditional regression testing, which showed that *additional* techniques generally perform better than *total* techniques in TCP [20, 48, 51, 79]. In summary, we are the first to find that the *additional* strategy is preferred over the *total* strategy even for configuration testing.

5.1.2 Comparing Coverage Criteria. From Table 3, we can observe that traditional code coverage at method granularity is still effective in test-case prioritization for configuration testing. For example, the *additional* code-coverage-based TCP techniques outperformed others in APMD, in which CC^{m}_{add} has the best performance. The reason is that a ctest with higher code coverage is more likely to exercise its covered configuration parameters in more project

components, and thus has a higher chance to detect potential misconfiguration(s). Moreover, configuration-specific coverage criteria can outperform traditional code coverage on APMDc. For example, the *additional* stack-trace-based TCP (ST_{add}) is in a statistically better group than CC^{m}_{add} in APMDc. The potential reason is that ctests with larger traditional code coverage also tend to run slower; in contrast, configuration-specific coverage can also effectively guide misconfiguration detection, but ctests with higher configuration-specific coverage do not necessarily run slower.

Among the configuration-specific coverage criteria, the best stack-trace-based TCP technique (ST_{add}) usually performs better than the best parameter-coverage-based TCP techniques (PC_{add}) on APMD and APMDc. The reason is that different ctests reading the same parameters may have greatly different invocation contexts and thus may have different capabilities in detecting misconfigurations. Another interesting finding is that both the best stack-trace-based and parameter-coverage-based techniques tend to outperform their change-aware counterparts. For example, ST_{add} achieves 0.895 (0.917) in APMDc (APMD), while ST^{D}_{add} has 0.877 (0.898). The reason is that majority of configuration changes are relatively small. Thus, the *additional* techniques cannot easily prioritize ctests with new change-aware configuration-specific coverage, and behave as random baseline when no ctests have new coverage.

5.1.3 IR-Based TCP. Although IR-based techniques (§3.1.1) have been recently claimed to be the state-of-the-art in test-case prioritization and unsafe selection for traditional regression testing [41, 51], they never perform the best in configuration testing on APMD and APMDc. There are several potential reasons. First, configuration changes are usually small and less informative than code changes. Second, unlike code changes, configuration changes have no surrounding context [41]. Thus, each change query is built simply from tokenized names of changed parameters (§4.4), which can often be too ambiguous. For example, the query built from changed parameters `{dataDir, dataLogDir}` is a bag of words `[data, dir, data, log, dir]`, which can be common in test files. Another interesting finding is that IR-based techniques never perform the worst in configuration testing. In fact, IR-based techniques are the most stable ones: in Figure 5, the plots for IR-based techniques are more concentrated near the median for both APMD and APMDc. The stability across runs for each project comes from test documents being large and diverse, so few ties are produced. Also, IR-based techniques prioritize ctests whose documents are more related to the names of changed parameters.

5.1.4 QTF-Based TCP. QTF has the second *highest* average APMDc, but the absolutely *lowest* average APMD across all projects. The reason is that a considerable portion of ctests are transformed from unit tests that have rather short execution time. Thus, QTF prioritizes these faster ctests first and can end up running many more ctests than other TCP techniques before detecting the misconfigurations, leading to low APMD values. However, when considering the test cost for APMDc, QTF is much more cost-effective, because the ctests prioritized earlier have short execution time. For example, on HDFS, many ctests prioritized earlier cost less than 0.1 second.

5.1.5 APMD vs. APMDc. While the rankings of many TCP techniques are similar by both APMD and APMDc, the diametrically opposite ranking of QTF when using APMD and APMDc indicates that APMD is *not* appropriate and can be misleading for configuration testing. This finding is consistent with prior work on traditional regression testing: APFD has been shown to be misleading in comparing TCP techniques because it does not consider test execution time [6, 30]. Therefore, in the following sections, we only focus on the APMDc results. Moreover, the high effectiveness of QTF in APMDc also inspired us to combine the basic techniques with test execution time information for hybrid techniques (§3.3).

Table 4: Per-project results for basic non-peer-based TCP

TCP	HCom.		HDFS		HBase		ZooK.		Alluxio	
	A.c	A.	A.c	A.	A.c	A.	A.c	A.	A.c	A.
ST_{add}	.990	.998	.608	.652	.971	.974	.963	.964	.941	.995
QTF	.998	.990	.865	.621	.997	.963	.909	.883	.679	.385
ST^{D}_{add}	.998	.999	.604	.646	.908	.928	.934	.923	.939	.996
CC^{m}_{add}	.990	.999	.587	.742	.871	.945	.993	.993	.935	.993
PC_{add}	.929	.992	.726	.739	.948	.948	.935	.928	.811	.807
PC^{D}_{add}	.995	.996	.685	.681	.947	.947	.917	.928	.807	.813
IR_{high}	.918	.895	.855	.887	.960	.980	.925	.925	.669	.803
IR_{low}	.934	.911	.876	.914	.904	.974	.909	.915	.672	.808
CC^{s}_{add}	.991	.998	.520	.718	.853	.922	.993	.993	.923	.986
Rand	.992	.993	.577	.578	.947	.946	.939	.938	.823	.820
PC^{D}_{tot}	.985	.993	.874	.868	.701	.685	.947	.961	.698	.837
PC_{tot}	.958	.979	.647	.567	.776	.762	.948	.947	.685	.799
CC^{m}_{tot}	.985	.992	.382	.660	.778	.793	.993	.993	.852	.906
CC^{s}_{tot}	.986	.992	.378	.658	.776	.793	.993	.993	.793	.892
ST_{tot}	.985	.985	.230	.308	.781	.779	.986	.991	.730	.866
ST^{D}_{tot}	.978	.992	.268	.342	.780	.774	.940	.917	.675	.860

5.1.6 Per-Project Results. Table 4 further presents the detailed average results for each studied project. The main findings—such as *additional* is better than *total*, and QTF is competitive—from the overall distribution of APMD/APMDc across all projects are also similar for individual projects. Thus, we do not show per-project results in the other RQs due to space limit and results being similar.

5.2 RQ2: Hybrid Non-Peer-Based TCP

This RQ evaluates the effectiveness of hybrid non-peer-based TCP techniques with two hybrid models discussed in §3.3. Figure 6 shows the distribution of APMDc values for each hybrid non-peer-based technique: the names of corresponding basic non-peer-based techniques are shown on the x-axis, while the green/orange violin plots show the distribution of APMDc values for divide-by-time (T_{div})/break-tie-by-time (T_{tie}) hybrid non-peer-based TCP techniques. Table 5 shows the overall average APMDc values and Tukey HSD groups for each TCP technique under the two hybrid models. Note that QTF+T_{div} serves as a baseline for T_{div} hybrid techniques—it is effectively Rand—while Rand+T_{tie} serves as a baseline for T_{tie} hybrid techniques—it is literally Rand.

5.2.1 Hybrid vs. Basic Non-peer-based TCP. Both hybrid models improved the average APMDc values across projects on most of the basic non-peer-based techniques. For example, excluding the baselines, the average APMDc values over all basic non-peer-based techniques is 0.838 (Table 3), while the same values for T_{tie} and

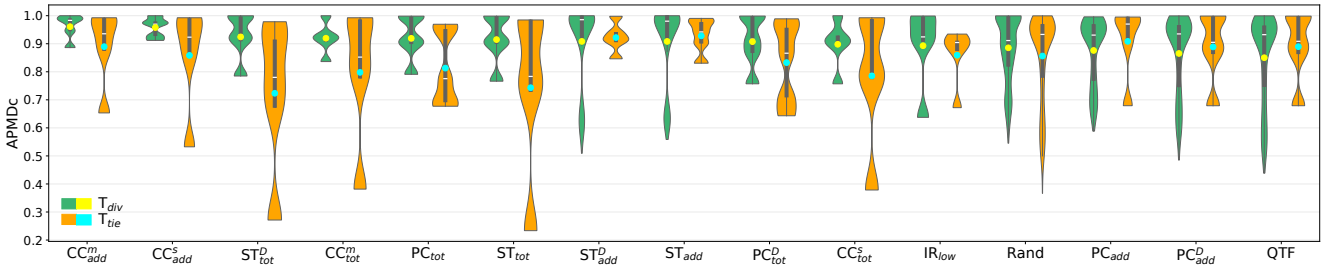

Figure 6: Distribution of APMDc values for hybrid non-peer-based TCP techniques (sorted by average APMDc from T_{div})

T_{div} hybrid techniques are 0.848 and 0.905, respectively. Also, the best basic non-peer-based technique (ST_{add}) achieves an APMDc value of 0.895, while the best hybrid non-peer-based technique, CC^m_{add}+T_{div}, achieves an APMDc of 0.961. CC^m_{add}+T_{div} performs much better than CC^m_{add}: CC^m_{add} favors ctests with larger code coverage but they also tend to run slower, while CC^m_{add}+T_{div} considers code coverage per time unit cost (§3.3.1), so CC^m_{add}+T_{div} makes a better trade-off between the coverage and cost information. In summary, this finding indicates that the hybrid models can substantially boost the basic non-peer-based TCP techniques. This finding was previously reported for traditional regression testing [41] but not for configuration testing.

Table 5: Results for hybrid non-peer-based TCP

TCP	Average		Group	
	T_{div}	T_{tie}	T_{div}	T_{tie}
CC^m_{add}	.961	.890	A	BC
CC^s_{add}	.958	.858	A	CD
ST^D_{tot}	.924	.723	B	G
CC^m_{tot}	.920	.798	BC	F
PC_{tot}	.919	.813	BC	EF
ST_{tot}	.915	.743	BCD	G
ST_{add}	.908	.928	BCDE	A
ST^D_{add}	.908	.922	BCDE	AB
PC^D_{tot}	.907	.833	BCDE	DE
CC^s_{tot}	.898	.785	CDEF	F
IR_{high}	.893	.865	DEF	CD
IR_{low}	.893	.859	DEF	CD
Rand	.886	.856	EFG	CD
PC_{add}	.876	.909	FG	AB
PC^D_{add}	.865	.889	GH	BC
QTF	.850	.890	H	BC

5.2.2 Divide-by-time vs. Break-tie-by-time. Table 5 shows that T_{div} hybrid techniques overall perform better than T_{tie} hybrid techniques. The average APMDc values range from 0.850 to 0.961 for T_{div}, while they range from 0.723 to 0.928 for T_{tie}. Interestingly, the *additional* TCP techniques with configuration-specific coverage tend to perform better with T_{tie} than with T_{div}, opposite to our overall finding. The reason is that ctests usually do not read as many (changed) configuration parameters as they cover traditional methods or statements; when using the *additional* strategy on configuration-specific coverage, the basic priority scores of ctests quickly become 0 (already prioritized ctests cover all parameters, and yet-to-prioritize ctests cannot cover any more parameters), thus making T_{div} effectively become random. For example, on HDFS, PC^D_{add} cannot provide *additional* coverage after prioritizing 2–4 ctests. In contrast, the T_{tie} hybrid model can break such ties by ordering the tied tests in the ascending order of their execution time (§3.3), thus outperforming T_{div} in such cases.

5.2.3 Total vs. Additional. With the T_{tie} model, the *additional* hybrid techniques outperform all the *total* ones on average APMDc values. This finding is consistent with our finding for the basic non-peer-based techniques in §5.1.1. Interestingly, this no longer holds for the T_{div} model. Although the very best T_{div} hybrid techniques (CC^m_{add}+T_{div} and CC^s_{add}+T_{div}) are *additional*, all other *additional* techniques under-perform their *total* counterparts with the T_{div} hybrid model. The reason is that the priority of ctests can easily become 0 when using the *additional* strategy, making T_{div} behave as random (§5.2.2), while the *total* strategy can still effectively prioritize different ctests. Thus, the T_{div} hybrid model is more effective for *total* TCP techniques that seldom encounter 0 priority scores. Also, T_{div} can be more effective for basic criteria that include more elements and are more diverse, such as traditional code coverage.

5.3 RQ3: Peer-Based TCP

This RQ evaluates the effectiveness of both basic (§3.2) and hybrid (§3.3) peer-based TCP techniques for configuration testing. Figure 7 shows the distribution of APMDc values for all the evaluated peer-based techniques. Table 6 further shows the average APMDc values and the Tukey HSD groups for these techniques.

5.3.1 Peer-Based vs. Non-Peer-Based TCP. According to Table 6, 7 of the 12 basic peer-based techniques outperform the *best* non-peer-based technique (i.e., CC^m_{add}+T_{div}) by average APMDc. Moreover, as seen in Figure 7, all APMDc values for all peer-based techniques are well above 0.65, while multiple basic and hybrid non-peer-based techniques have APMDc values well below 0.65 even up to 0.2 (Figure 5 and Figure 6), indicating the effectiveness and stability of the basic peer-based techniques for configuration testing.

$Para^{PC}_{add}$ and $Para^{RC}_{add}$ are statistically significantly better than other basic peer-based techniques, as they are both within the best Tukey HSD group "A". These two techniques are not statistically different, although $Para^{PC}_{add}$ has a slightly higher average APMDc. This finding is surprising as $Para^{PC}_{add}$ requires *no* root-cause information, but still performs as well as $Para^{RC}_{add}$, which requires such information (§3.2). The reason is that on some projects (e.g., ZooKeeper), many ctests have similar $Para^{RC}$, so the *additional* strategy suffers the same problem as in §5.2.3. Meanwhile, $Para^{PC}$ values of these ctests are more diverse (and larger than their $Para^{RC}$ values).

Different from the results for the non-peer-based techniques, the hybrid models have only limited effectiveness for the peer-based techniques. The T_{div} model can only improve the effectiveness for the inferior peer-based techniques. For example, $Conf^{all}_{tot}$, the worst basic technique, is improved from 0.899 into 0.945, while the two best basic techniques ($Para^{PC}_{add}$ and $Para^{RC}_{add}$) have almost no change.

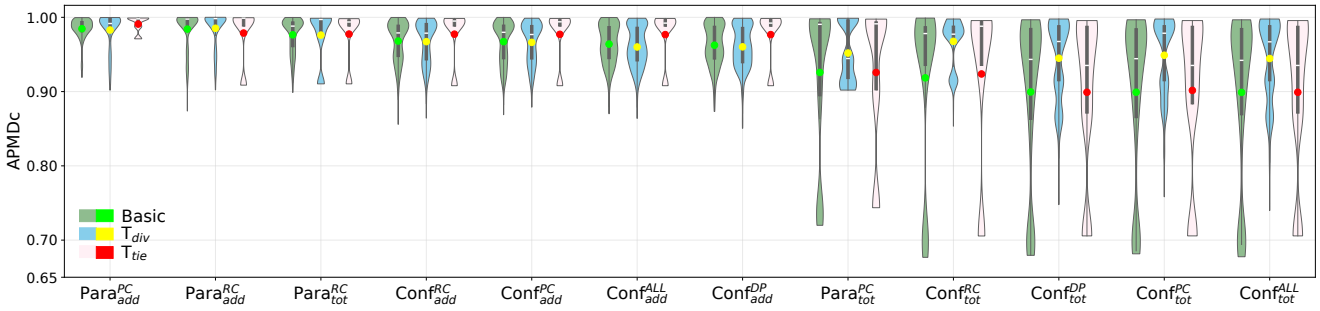

Figure 7: Distribution of APMDc values for peer-based TCP techniques (sorted by average APMDc from Basic)

Table 6: Results for peer-based TCP techniques

TCP	Average			Group		
	Basic	T_{div}	T_{tie}	Basic	T_{div}	T_{tie}
$Para^{PC}_{add}$	.985	.983	.991	A	AB	A
$Para^{RC}_{add}$	.984	.985	.979	A	A	A
$Para^{RC}_{tot}$	.976	.976	.977	AB	B	A
$Conf^{RC}_{add}$	.968	.967	.977	B	C	A
$Conf^{PC}_{add}$	.967	.966	.977	B	C	A
$Conf^{all}_{add}$	.964	.960	.977	B	C	A
$Conf^{DP}_{add}$	.962	.960	.977	B	C	A
$Para^{PC}_{tot}$	.926	.952	.926	C	D	B
$Conf^{RC}_{tot}$	.918	.968	.924	C	C	B
$Conf^{PC}_{tot}$	.899	.949	.902	D	D	C
$Conf^{all}_{tot}$	.899	.945	.899	D	D	C
$Conf^{DP}_{tot}$	.899	.945	.899	D	D	C

The T_{tie} model can only slightly improve the effectiveness of the superior peer-based techniques. For example, $Para^{PC}_{add}$ changes from 0.985 to 0.991, while the inferior techniques (such as $Conf^{all}_{tot}$) do not change at all. The reason is that *total* techniques usually have fewer ties, making T_{div} more effective than T_{tie}.

5.3.2 Configuration vs. Parameter Granularity. Using both *additional* and *total* strategies, techniques at the parameter granularity have mostly outperformed techniques at the configuration granularity. For example, as seen from Table 6, with the *additional* strategy, the basic techniques at the parameter granularity ($Para^{PC}$, $Para^{RC}$) are both in group "A", while all basic techniques at the configuration granularity ($Conf^{all}$, $Conf^{DP}$, $Conf^{PC}$, $Conf^{RC}$) are in group "B". Similarly, with the *total* strategy, the basic $Para^{RC}$and $Para^{PC}$ are in groups "AB" and "C", respectively, while all basic techniques at configuration granularity are within groups "C" or "D". This result is expected as the parameter granularity captures parameter-level information from other failed peer configuration changes, while the configuration granularity is more coarse-grained (§3.2.2).

5.3.3 Total vs. Additional. Similar to the results for the non-peer-based techniques, the *additional* strategy generally performs better than the *total* strategy for the basic and T_{tie} peer-based techniques. Except that Table 6 shows the basic $Para^{RC}_{tot}$ is a *total* technique at the parameter granularity that performed slightly better than basic *additional* techniques at the configuration granularity, because $Para^{RC}$ leverages more fine-grained information about peer misconfigured parameters to guide more effective prioritization.

Table 7: Results for the best TCP techniques

TCP	HCom.	HDFS	HBase	ZooK.	Alluxio	Avg	Group
$Para^{PC}_{add}$+T_{tie}	.999	.988	.999	.971	.998	.991	A
$Para^{PC}_{add}$	.995	.982	.990	.975	.981	.985	A
CC^{m}_{add}+T_{div}	1.00	.886	.989	.946	.983	.961	B
ST_{add}	.990	.608	.971	.963	.941	.895	C
QTF	.998	.865	.997	.909	.679	.890	C
Rand	.992	.577	.947	.939	.823	.856	D

5.4 Summary

We compare the best techniques from each of the basic/hybrid peer-based/non-peer-based categories, i.e., ST_{add} (basic non-peer-based), CC^{m}_{add}+T_{div} (hybrid non-peer-based), $Para^{PC}_{add}$ (basic peer-based), and $Para^{PC}_{add}$+T_{tie} (hybrid peer-based). We also include Rand and QTF as the baselines. Note that the QTF technique is rather competitive as it outperforms almost all the basic non-peer-based TCP techniques (Table 3). Table 7 presents the main comparison results. We can observe that all four techniques significantly outperform the Rand baseline, and three of them significantly outperform the QTF baseline. In summary: (1) CC^{m}_{add}+T_{div} is the best non-peer-based technique and recommended when no peer configuration information is available, (2) $Para^{PC}_{add}$ and its T_{tie} counterpart are the best techniques (i.e., both in group "A") and recommended when peer configuration information is available.

5.5 Threats to Validity

External validity. The threats to external validity mainly lie in projects and dataset used in this work. To reduce such threats, we directly use all the real-world projects and configuration changes from the Ctest dataset [36]. However, our evaluation is only based on ctests, which cannot represent all possible types of configuration tests. Future work should consider more diverse datasets and other types of configuration tests.

Internal validity. The threats to internal validity mainly lie in the potential bugs in our techniques and experimental scripts. To reduce such threats, the authors regularly check the results and

code to eliminate potential bugs. Furthermore, we released all our dataset and code to benefit the community.

Construct validity. The threats to construct validity mainly lie in the metrics used in our study. To reduce such threats, we adapt two widely-used metrics for evaluating TCP techniques (APFD and its cost-aware variant APFDc) and propose new metrics (APMD and its cost-aware variant APMDc) for configuration testing.

6 DISCUSSION AND FUTURE WORK

To better measure the overall detection time for all the misconfigured parameters within each configuration change, we introduced APMDc (together with APMD) as our main evaluation metric. However, APMDc may not be preferred for practitioners with more interest in how TCP affects the time to detect misconfigurations. Thus, we also relate changes to APMDc with changes to the total test time. APMDc captures time to detect *all* misconfigured parameters in a configuration change. If there is only one misconfigured parameter, then 0.1 increase in APMDc maps to exactly 10% reduction of total time. If there are more misconfigured parameters, 0.1 may map to less or more than 10% time reduction to detect either the first misconfigured parameter or all misconfigured parameters. For our studied projects, 0.1 increase in APMDc maps to from 7.86% (HCommon) to 21.93% (HBase) average time reduction to detect all misconfigured parameters. The reduction can be even larger to find the first misconfigured parameter, e.g., 0.1 increase in APMDc maps to 53.38% (Alluxio) average time reduction.

Our study also points to several directions for future work. Since historical data were reported to be useful in traditional test-case prioritization [10, 25, 41, 50], we could leverage historical configuration change test results from earlier *code* versions to develop history-based TCP techniques for configuration testing. We also consider improving the current configuration-specific TCP techniques and evaluating them on larger datasets. For example, we can fuse deeper context information (e.g., how ctests *use* their parameters acquired from configuration taint analysis) into stack-trace-based TCP techniques, or improve peer-based TCP techniques by combining more data from peer configurations (e.g., test time, failure stack traces).

Furthermore, we plan to understand the impact of software evolution on the performance of our evaluated TCP techniques for configuration testing. Although prior work has shown that the traditional prioritization techniques remain robust over multiple system releases [15], this conclusion may not hold in the context of configuration testing. Configurations and configuration-related code are updated frequently [60, 84], so certain types of test information may be more sensitive to software evolution. For example, data from old peer configuration changes could be less accurate in guiding peer-based TCP techniques on recent system releases.

We only evaluate the performance of TCP techniques on configuration changes. However, sometimes software developers may change both configuration and code in the same commit. In such context, a TCP technique should consider both configuration and code information, and balance the effectiveness in speeding up both misconfiguration and code fault detection. We plan to study how the mixture of configuration and code testing can shift the performance of our evaluated TCP techniques, and understand how to develop competitive TCP techniques in such context.

7 RELATED WORK

We have already introduced the background on configuration testing (§2) and discussed the related test-case prioritization (TCP) techniques (§3), so this section briefly discusses the basics and applications of TCP. TCP techniques were initially proposed to reorder test executions for traditional software systems (e.g., common C and Java applications) to speed up detection of regression faults during software evolution. To date, a large number of code-coverage-based TCP techniques have been proposed for such purpose, including techniques based on traditional *total*/*additional* heuristics [28], adaptive random testing [20], genetic algorithms [26], and constraint solving [80]. More recently, researchers have also looked into TCP techniques that do not require code-coverage information, e.g., techniques based on information retrieval [41] or static program analysis [29]. Interestingly, although more and more TCP techniques have been proposed, the traditional *additional* technique and its cost-cognizant variant (e.g., hybrid with divide-by-time) have still remained among the most effective TCP techniques [6].

Besides the traditional application scenarios, TCP has also been applied to various other scenarios, e.g., mutation testing [81], fault localization [13], and automated program repair [12, 27, 43]. Moreover, researchers have applied TCP techniques for testing configurable systems [44, 57]. However, they still target the traditional regression testing problem, i.e., detecting regression faults caused by code changes, while also considering prioritizing the potential configurations that may likely expose regression faults. In contrast, this paper makes the first attempt to apply TCP for speeding up misconfiguration detection for configuration testing.

8 CONCLUSION

We have performed the first extensive study of TCP for configuration testing. We have implemented 84 traditional and novel ctest-specific TCP techniques. The experimental results on five popular cloud projects demonstrate that TCP can substantially speed up misconfiguration detection. We have also analyzed the impact of various controllable factors for applying TCP in configuration testing, including coverage criteria, hybrid models, *total*/*additional* strategies, peer-data granularities, and study metrics. In sum, our study reveals various practical guidelines for applying TCP in configuration testing, including: (1) among the basic TCP techniques, QTF is surprisingly competitive and often outperforms sophisticated techniques (based on code coverage or IR) and even some ctest-specific techniques (based on parameter coverage or stack traces), (2) hybrid TCP techniques (which enhance basic techniques with text execution cost information) can boost the performance of most basic techniques, and (3) peer-based TCP techniques (which leverage peer configuration data for better prioritization) can substantially outperform all other studied TCP techniques.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by NSF grants CCF-1763788, 1763906, 1816615, 1942430, 2029049, and CNS-1740916, 1956007. We also acknowledge support for research on regression testing from Facebook, Futurewei, and Google; a Facebook Distributed Systems Research award; Microsoft Azure credits; and Google Cloud credits.

REFERENCES

[1] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *OSDI*.

[2] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *OSDI*.

[3] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan and Claypool Publishers.

[4] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud. In *Middleware*.

[5] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *ICSE*.

[6] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing Test Prioritization via Test Distribution Analysis. In *ESEC/FSE*.

[7] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *ESEC/FSE*.

[8] Docker Hub 2020. Docker Hub. https://www.docker.com/products/docker-hub.

[9] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *ICSE*.

[10] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE*.

[11] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical Evaluation of Pareto Efficient Multi-Objective Regression Test Case Prioritisation. In *ISSTA*.

[12] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA*.

[13] Alberto Gonzalez-Sanchez, Eric Piel, Hans-Gerhard Gross, and Arjan JC van Gemund. 2010. Prioritizing Tests for Software Fault Localization. In *QSIC*.

[14] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *SoCC*.

[15] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-Box and Black-Box Test Prioritization. In *ICSE*.

[16] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *TSE* 40, 7 (2014).

[17] Peng Huang, William J. Bolosky, Abhishek Sigh, and Yuanyuan Zhou. 2015. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *EuroSys*.

[18] Qiang Huang, Helen J. Wang, and Nikita Borisov. 2005. Privacy-Preserving Friends Troubleshooting Network. In *NDSS*.

[19] JavaParser 2020. JavaParser. https://javaparser.org/about.html.

[20] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. 2009. Adaptive Random Test Case Prioritization. In *ASE*.

[21] Robert Johnson. 2010. More Details on Today's Outage. http://www.facebook.com/note.php?note_id=431441338919.

[22] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *DSN*.

[23] Stuart Kendrick. 2012. What Takes Us Down? *USENIX ;login:* 37, 5 (2012).

[24] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo D'Amorim. 2013. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/FSE*.

[25] Jung-Min Kim and Adam Porter. 2002. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. In *ICSE*.

[26] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *TSE* 33, 4 (2007).

[27] Yiling Lou, Samuel Benton, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. How Does Regression Test Selection Affect Program Repair? An Extensive Study on 2 Million Patches. *arXiv:2105.07311* (2021).

[28] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-World Software Evolution?. In *ICSE*.

[29] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. 2018. How do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects. *TSE* 45, 11 (2018).

[30] Alexey G. Malishevsky, Joseph R Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. *Cost-Cognizant Test Case Prioritization*. Technical Report. TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln.

[31] Ben Maurer. 2015. Fail at Scale: Reliability in the Face of Rapid Change. *CACM* 58, 11 (2015).

[32] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *ICSE*.

[33] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, B. Ashok, Chetan Bansal, Chandra Maddila, Christian Bird, Sumit Asthana, and Aditya Kumar. 2020. Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis. In *NSDI*.

[34] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *ASE*.

[35] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2004. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI*.

[36] openctest 2020. openctest. https://github.com/xlab-uiuc/openctest.

[37] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done About It?. In *USITS*.

[38] Noam Palatin, Arie Leizarowitz, Assaf Schuster, and Ran Wolff. 2006. Mining for Misconfigured Machines in Grid Systems. In *KDD*.

[39] Marek Parfianowicz and Grzegorz Lewandowski. 2017–2018. OpenClover. https://openclover.org.

[40] David Paterson, José Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. 2019. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In *ICST*.

[41] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *ISSTA*.

[42] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. 2015. ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection. In *VLDB*.

[43] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair Through Fault-Recorded Testing Prioritization. In *ICSM*.

[44] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. 2008. Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization. In *ISSTA*.

[45] Ariel Rabkin and Randy Katz. 2011. Precomputing Possible Configuration Error Diagnosis. In *ASE*.

[46] Ariel Rabkin and Randy Katz. 2013. How Hadoop Clusters Break. *IEEE Software* 30, 4 (2013).

[47] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. 2000. Experimentation as a Way of Life: Okapi at TREC. *Inf. Process. Manag.* 36, 1 (2000).

[48] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*.

[49] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing Test Cases for Regression Testing. *TSE* 27, 10 (2001).

[50] David Saff and Michael D. Ernst. 2003. Reducing Wasted Development Time via Continuous Testing. In *ISSRE*.

[51] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *ICSE*.

[52] Gerard Salton and Christopher Buckley. 1988. Term-Weighting Approaches in Automatic Text Retrieval. *Inf. Process. Manag.* 24, 5 (1988).

[53] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing Configuration File Specifications with Association Rule Learning. In *OOPSLA*.

[54] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic Automated Language Learning for Configuration Files. In *CAV*.

[55] Alex Sherman, Phil Lisiecki, Andy Berkheimer, and Joel Wein. 2005. ACMS: Akamai Configuration Management System. In *NSDI*.

[56] Jonathan Shieber. 2019. Facebook Blames a Server Configuration Change for Yesterday's Outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage.

[57] Hema Srikanth, Myra B Cohen, and Xiao Qu. 2009. Reducing Field Failures in System Configurable Software: Cost-Based Prioritization. In *ISSRE*.

[58] M. Stone. 1974. Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society. Series B (Methodological)* 36, 2 (1974).

[59] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *OSDI*.

[60] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *SOSP*.

[61] John W. Tukey. 1949. Comparing Individual Means in the Analysis of Variance. *Biometrics* 5, 2 (1949).

[62] Ozan Tuncer, Nilton Bila, Canturk Isci, and Ayse K. Coskun. 2018. *ConfEx: An Analytics Framework for Text-Based Software Configurations in the Cloud*. Technical Report RC25675 (WAT1803-107). IBM Research.

[63] Helen J. Wang, Yih-Chun Hu, Chun Yuan, Zheng Zhang, and Yi-Min Wang. 2004. Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting. In *IPTPS*.

[64] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI*.
[65] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. 2004. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*.
[66] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations. In *USENIX ATC*.
[67] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. 2019. Towards Continuous Access Control Validation and Forensics. In *CCS*.
[68] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software. In *ESEC/FSE*.
[69] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *OSDI*.
[70] Tianyin Xu and Owolabi Legunsen. 2019. Configuration Testing: Testing Configuration Values as Code and with Code. *arXiv:1905.12195* (2019).
[71] Tianyin Xu and Darko Marinov. 2018. Mining Container Image Repositories for Software Configurations and Beyond. In *ICSE*.
[72] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *SOSP*.
[73] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Comput. Surv.* 47, 4 (2015).
[74] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *SOSP*.
[75] Shin Yoo and Mark Harman. 2012. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *STVR* 22, 2 (2012).
[76] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based Online Configuration Error Detection. In *USENIX ATC*.
[77] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *ESEC/FSE*.
[78] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *ASPLOS*.
[79] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the Gap between the Total and Additional Test-Case Prioritization Strategies. In *ICSE*.
[80] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-Aware Test-Case Prioritization using Integer Linear Programming. In *ISSTA*.
[81] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster Mutation Testing Inspired by Test Prioritization and Reduction. In *ISSTA*.
[82] Sai Zhang and Michael D. Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *ICSE*.
[83] Sai Zhang and Michael D. Ernst. 2014. Which Configuration Option Should I Change?. In *ICSE*.
[84] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In *ICSE*.