CTForge: Automatically Generating Test Suites for Software Configuration

Yuanliang Zhang, Zhizheng Zheng, Shanshan Li, Zhouyang Jia, Chaopeng Luo, Liqian Chen, Zhenbang Chen, Ji Wang, Xiangke Liao
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
{zhangyuanliang13,zhengzhizheng23,shanshanli,jiazhouyang,luochaopeng18,chenliqian2001,zbchen,wj,xkliao}@nudt.edu.cn

Abstract

Software configuration plays a critical role in modern software systems, yet it also frequently leads to severe failures. Existing approaches, whether static analysis or LLM-based validation, struggle to capture implicit semantic constraints and validate configuration-code interactions. Leveraging existing unit tests is also insufficient due to their lack of configuration-specific focus. Thus, we present CTForge, an LLM-powered framework that automatically generates configuration-specific test suites. Its core innovation is a three-stage configuration-aware optimization pipeline that moves beyond vanilla test generation, specifically using parameter semantics to improve LLM outputs. This optimization pipeline iteratively aligns tests through: (1) enhance configuration-related code coverage to ensure relevance, (2) assertion validation across valid parameter values to eliminate false assertions, and (3) configuration-sensitivity verification to select tests that semantically exercise the intended configuration behavior. Evaluated on five large-scale systems, CTForge detects 63 out of 79 real-world configuration issues and achieves an F1-score of 0.88 in misconfiguration detection. CTForge also uncovers six previously unknown defects. Our work demonstrates that structured configuration-aware refinement is essential for LLM to produce effective test suites.

CCS Concepts

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

Keywords

Configuration Testing; Test Generation; Large Language Models

ACM Reference Format:
Yuanliang Zhang, Zhizheng Zheng, Shanshan Li, Zhouyang Jia, Chaopeng Luo, Liqian Chen, Zhenbang Chen, Ji Wang, Xiangke Liao. 2026. CTForge: Automatically Generating Test Suites for Software Configuration. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837484

Yuanliang Zhang, Shanshan Li, Zhouyang Jia, Zhenbang Chen and Ji Wang are also with the State Key Laboratory of Complex & Critical Software Environment.
Zhizheng Zheng contributes equally to the paper.
Shanshan Li is the corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837484

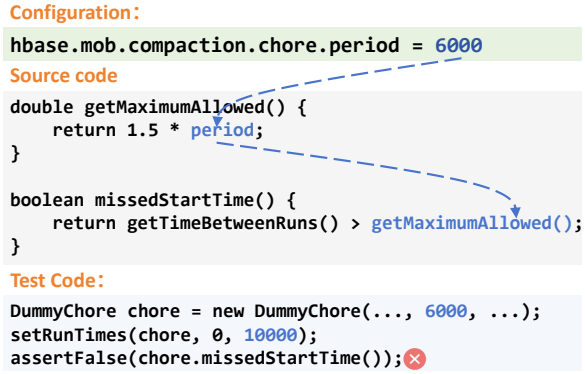


Figure 1: A bug example exposed by configuration test.

1 Introduction

Configuration serves as a critical mechanism for controlling functionality and resource allocation in modern software systems. However, configuration-induced issues frequently lead to software failures [72, 73, 78]. Studies show that configuration-induced incidents account for a significant portion of service disruptions in large-scale cloud systems [58, 62]. These errors often arise from subtle violations of hidden constraints or dormant code bugs exposed by valid configuration values [60], underscoring the urgent need for robust configuration validation before deployment.

Some existing work checks whether the values of configuration parameter violate certain constraints. These constraints are extracted by static analysis of source code [40, 72] or synthesized by large language models [39]. However, the former can only extract specific types of constraints, while the latter is plagued by false positives caused by LLM hallucination. To address these problems, today's configuration management systems employ continuous configuration testing [62]. One attempt [60] is to leverage existing unit tests to assess configuration changes in context, but its effectiveness is essentially bounded by the original quality of existing tests. Since most unit tests prioritize functional correctness over configuration-specific scenarios, the critical Configuration-Related Code (CR Code) often remains untested. This limitation motivates our research on generating high quality configuration-specific test suites. Fig 1 illustrates an example that a test can reveal a configuration-semantic bug [22]. The parameter `hbase.mob.compaction.chore.period` is propagated to `getMaximumAllowed()`, and further affects the decision made by `missedStartTime()`. The time unit millisecond is used by `getTimeBetweenRuns()`, if `period` uses another unit, the service will always miss its start time. The test exposes this bug through the assertion of miss start time behavior.

Generating high quality configuration tests for large-scale software is challenging. First, these systems involve complex cross-component interactions that transcend granularity at the method-level targeted by traditional unit test generation. Second, generated tests are difficult to validate: failing tests may indicate defects in either the system or the test code itself, while passing tests may be ineffective due to weak or irrelevant assertions. Relying solely on code coverage metrics is insufficient as they do not capture configuration-specific semantics. Recently, LLM-based test generation has demonstrated the potential to generate semantically meaningful tests [15–17, 20, 44, 56, 59, 67, 74]. However, LLM-generated tests remain limited by a lack of domain-specific knowledge and are prone to hallucinations [25, 79]. As a result, LLMs may produce incorrect oracles. More critically, they can generate tests that appear plausible but are ineffective, which fail to detect configuration-related issues, but increase testing overhead.

We present CTForge, a framework that generates configuration-specific tests. CTForge first employs taint analysis to extract configuration contexts and constraints, providing rich semantics for LLM. **Beyond leveraging the LLM's raw test-generation capacity, the core contribution of CTForge is a three-stage, configuration-semantics-aware optimization pipeline, which distinguishes our approach from vanilla LLM-based test generation.** This pipeline acts as an iterative refinement loop that: 1) ensures tests cover critical CR Code, 2) validates test assertions against a spectrum of valid parameter inputs, and 3) verifies test sensitivity to configuration semantics, ensuring they fail when the intended configuration logic is altered.

We evaluate CTForge on five large scale software systems (HCommon, HDFS, HBase, Alluxio, and ZooKeeper). We collect 79 real-world configuration-related issues with different types from existing datasets [60, 78]. CTForge identifies 63 issues in total, outperforming existing configuration testing and validation approaches. In addition, we inject 1160 values for the 232 parameters to explore a wider configuration space. CTForge detects on average 79% of the injected error values and achieves the highest F1 score (0.88). During our experiment, we find six new bugs. Finally, we conduct an ablation study to assess how much each optimization stage improves upon vanilla LLM-based test generation.

In general, this paper makes the following contributions.

- We propose a configuration-specific optimization pipeline over vanilla LLM-based test generation. By leveraging iterative semantic-guided optimization, it produces tests that target constraints and behavior at parameter level.
- We implement CTForge, an extensible framework to generate configuration tests. We release the code and dataset for future research in this area.
- We demonstrate CTForge's effectiveness on five widely-used software systems. The results show that CTForge outperforms state-of-the-art configuration validation techniques and reveals six previously unknown bugs.

2 Background

2.1 How Does Configuration Break the System?

Configuration-induced issues arise primarily from the configuration values and the underlying code, which manifests itself through two interconnected mechanisms: invalid parameter values and defective configuration code. Invalid parameter values may cause failures by violating explicit constraints. Although static analysis can detect basic constraints such as type errors or values range violations [11, 52, 72, 76], it may fails to identify semantic inconsistencies caused by hidden dependencies or environmental factors [26, 60]. Similarly, defects in configuration-related code can lead to severe failures when interacting with specific configuration values [37, 38]. Flaws in areas such as configuration parsing logic, resource handling, or API implementations may misinterpret or improperly handle certain values—even those that are syntactically correct [55, 78]. More subtly, seemingly correct values may activate untested code paths or operational modes—such as enabling certain features or scaling options, that expose dormant defects in the code [71]. Such defects often go undetected because conventional testing tends to prioritize functional correctness over coherence between configuration values and code behavior, thus leaving inadequate tests for software configuration [38, 71].

2.2 Challenges and Opportunities

The complexity of configuration–code interactions presents challenges for automated testing.

- **Challenge 1:** Compared to general tests, how can we make tests more specific to the configuration code?
- **Challenge 2:** It is difficult to assess the correctness of the generated tests. It remains ambiguous whether a failing test reveals a real defect or an incorrect assertion synthesized by LLMs.
- **Challenge 3:** A purely generative approach tends to produce ineffective tests that are semantically shallow [66, 74], failing to target configuration failures.

However, these challenges create clear opportunities for a more integrated and feedback-driven methodology. Moving beyond raw LLM generation, we can design pipelines that ground test synthesis in actual configuration usage traces and refine output through iterative execution-based validation (§ 3.3). Such an approach can leverage configuration's unique characteristics for improving test relevance, assertion correctness, and semantic sensitivity. This synergy can present a promising path toward automatically creating high-quality configuration-specific tests.

3 Design of CTForge

We design CTForge, an automatic configuration-specific test suites generation framework, as shown in Fig 2. The framework operates in two integrated phases: Configuration Knowledge Retrieval and Configuration Test Generation. The first phase extracts configuration semantics from docs and source code (§ 3.1). The test generation phase then prompts the LLM to synthesize initial test candidates (§ 3.2) and employs a configuration-aware optimization pipeline (§ 3.3) to iteratively filter and refine these candidate tests.

3.1 Configuration Knowledge Retrieval

Directly prompting LLMs to generate configuration-specific tests performs poorly in practice. This difficulty arises from the inherent complexity of configuration usage in real-world systems. To address this challenge, we explore two complementary approaches

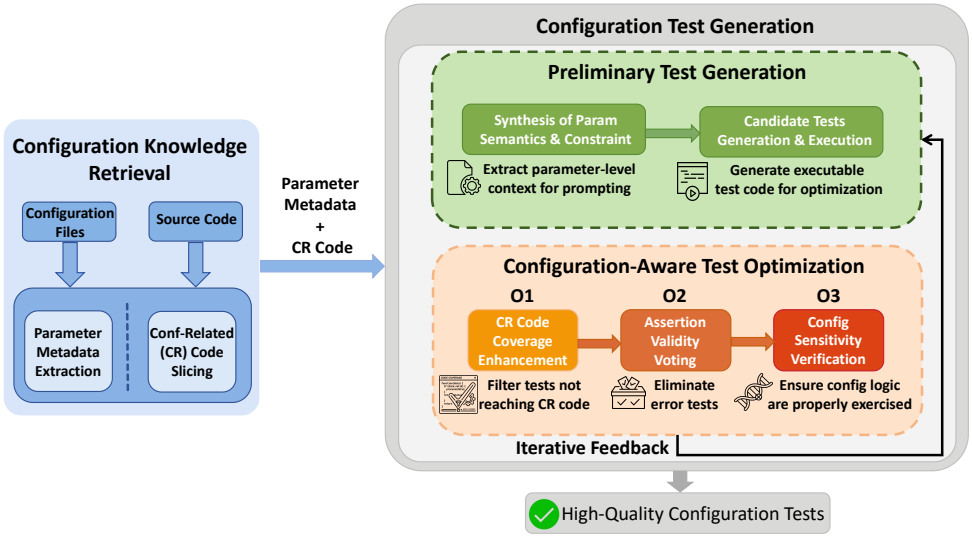


Figure 2: Overview of CTForge.

for supplying parameter knowledge to the LLM: 1) Extracting parameter metadata from documentation, and 2) Performing taint analysis on source code to obtain Configuration-related code (CR Code). We aim to bridge the gap between general-purpose LLM capabilities and parameter-specific knowledge. Finally, we construct the information as context to prompt LLM, as shown in Figure 3.

3.1.1 Parameter Metadata. Parameter metadata is extracted by parsing structured configuration specification documents that are commonly maintained as part of mature software systems [78]. In practice, parameters are conventionally recorded in formatted file (e.g., XML, JSON, YAML) adhering to a regular schema. We use a lightweight parser combining regex-based templates and DOM traversal to match tags such as <name>, <value>, and <description>, thereby extracting the parameter name, default value, and description text. The description fields often encode implicit constraints (e.g., value ranges, type expectations), intended behavioral effects (e.g., enabling or disabling specific features) [7, 30, 68]. They reflect important domain knowledge about how configuration parameters are expected to interact with the system at runtime.

3.1.2 Config-related Code (CR Code). In mature software systems, parameter values are typically parsed through key-value access interfaces and then passed through multiple layers of abstraction before reaching their final usage points [52, 78]. This propagation obscures the semantic connection between configuration parameters and their runtime behavioral impact, making it difficult for LLMs to infer from raw code alone.

We follow the definition of configuration-related code in the previous study [64, 78]. Parameter values are retrieved via APIs and assigned to local variables or stored in class fields. Then they will transmitted through method arguments or constructors, and eventually used in decision or computation logic. We apply taint analysis by treating parameter values as taint sources and tracing

Please design multiple test cases based on the Parameter Metadata and CR Code Snippet.

```
# Parameter Metadata
name: hbase.bulkload.retries.number
default: 10
constraint: integer >= 0
description: max retries for atomic bulk load; 0 means
unlimited
```

```
# CR Code Snippet
```java
int maxRetries = conf.getInt(···, 10);
        ···
if (maxRetries != 0 && count >= maxRetries) {
     throw new IOException(···);
}
```
```

```
# Return Template
```json
[{test_case_name, objective, steps, expected_result}
{test_case_name, objective, steps, expected_result}
···]
```
```

Figure 3: Example of context construction for LLM prompt.

how parameter values propagate through the program to their downstream usage points [2, 64]. The analysis finally extracts a parameter-specific code slice from the whole code base.

3.2 Preliminary Test Generation

CTForge first generates test candidates by: 1) decoupling configuration parameter with test logic from fixed values, making tests adaptable to any runtime configuration values, 2) leveraging a Two-Tier Testing Strategy that validates both value correctness and runtime behavioral impact.

3.2.1 Configuration Decoupling. To faithfully reflect how configurations are used in practice, we deliberately decouple parameter

```java
@Test
public void testMaxRetries() {
    Configuration conf;

    int bulkLoadRetries =
conf.getInt("hbase.bulkload.retries.numbe
r");
    ...
    assertTrue("retries must be >= 0",
bulkLoadRetries  >= 0);
}
```

(a) value validity test

```java
@Test
public void testCryptoBufferSize() {
    int maxThreads = conf.getInt("···");

    ThreadPoolExecutor pool =
   new ThreadPoolExecutor(···,
maxThreads);

    assertEquals(maxThreads,pool.getMaxim
umPoolSize());
}
```

(b) config functional test (data-type)

```java
@Test
public void testMaxRetries() {
    Configuration conf;

    conf.setBoolean("hadoop.security.auth
orization", true);

    startRpcServer(conf);

    assertTrue(authorizationEnabled());
}
```

(c) config functional test (control-type)

Figure 4: Examples of generated configuration-specific tests.

values from test logic. Rather than hard-coding specific values in test setup phase, we synthesize tests that retrieve parameter values from configuration file at runtime. This design ensures that the generated tests are not bound to a fixed parameter value but instead adapt to the actual settings, reflecting real deployment scenarios and improving the flexibility and maintainability of the test suite to test more parameter values.

3.2.2 Two-Tier Configuration Test. CTForge synthesizes targeted tests through a two-tier strategy designed to validate both the validity of parameter values and their behavioral impact on the system. This strategy produces two complementary forms of tests.

Value Validity Testing, as shown in Fig. 4 (a), establishes a baseline of validity for parameter values themselves. CTForge leverages the collected parameter knowledge (type, range, dependencies) to prompt the LLM to generate tests that assert whether a given value complies with its documented or inferred constraints. For example, check that a numeric parameter is positive or that a string parameter matches an expected pattern. While this tier effectively eliminates syntactically invalid or out-of-bound values, it operates purely on the input space and does not verify how those values are actually used by the CR Code at runtime.

Config Functional Testing addresses the fundamental question: Does this configuration setting cause the system to behave as intended? Since configuration usage is often buried deep within methods (including private ones), we use the previously identified CR Code paths as focal points for test generation. Following the definition in [55], we differentiate two parameter types: 1) data-type parameters (e.g., buffer sizes, timeouts), of which the tests verify that the runtime value (e.g., the actual buffer allocated) matches the configured setting, as shown in Fig. 4 (b), 2) control-type parameters (e.g., boolean flags), of which the tests verify that the corresponding functionality is correctly enabled/disabled and that the associated code paths are executed, as shown in Fig. 4 (c). This tier ensures that the parameter values correctly enact their intended semantic effect on the system, bridging the gap between a static value and its dynamic behavioral footprint.

3.3 Configuration-Aware Test Optimization

The preliminary tests generated by LLMs suffer from irrelevance, wrong assertions, and a lack of configuration-specific focus. To systematically convert these raw outputs into high-quality configuration tests, CTForge employs a three-stage optimization pipeline followed by three design guidlines. This pipeline operates as an iterative filter and enhancer, with each stage designed to address a specific class of deficiencies. In practice, we set the iteration number to 6, which is general in existing LLM generation tasks [74].

3.3.1 CR Code Coverage Enhancement. A test achieving high overall coverage may still neglect the specific code paths that parse, propagate, and use configuration values, which are the primary targets for validation. Therefore, we establish CR code reachability as a prerequisite. Operationally, for each compilable test candidate, the framework executes it, collecting standard line and branch coverage. Crucially, this coverage report is then aligned with the statically identified CR code segments. A test is filtered out if it does not touch any CR code. If target CR code locations remain uncovered, the mechanism provides targeted feedback, such as the uncovered code spans and their calling contexts to guide adjustments in test setup or invocation before re-entry into the refinement loop.

> **Design Guideline for Challenge 1:** Code coverage metrics should be specifically applied to configuration-related code rather than the whole code base.

3.3.2 Assertion Validity Voting. Coverage refinement ensures that generated tests reach CR Code, but it cannot filter out tests with defective assertions, a well-known challenge in automated test generation [24, 56]. The core difficulty lies in the test oracle problem: when a test fails, it is fundamentally ambiguous whether the failure exposes a bug in the code under test or stems from an incorrect assertion synthesized by the generator. This issue is distinct from flaky tests; rerunning the test is insufficient because the root cause is not non-determinism but a semantic mismatch between the test's expectation and the system's intended behavior.

Configuration testing presents a unique opportunity to mitigate this oracle ambiguity. A test that frequently fails when provided with valid parameter values is likely to assert incorrect constraints and not reveal true defects. Based on this guideline, we design an assertion validity voting procedure, as shown in Algorithm 1. For each test, we execute it multiple times using different valid values as input (5 in our design). If all executions succeed, the test is treated as non-erroneous and retained for subsequent refinement. In contrast, if more than half of the executions fail, the test is classified as erroneous and discarded. When only a subset of executions fail, the test is identified as exposing a potential configuration defect.

Algorithm 1 Assertion Validity Voting

Input: Candidate test set T, Valid-value generator $GenValid(p)$ for parameter-under-test p, Voting budget k
Output: Retained tests T_{keep}, Discarded tests T_{drop}, Potential defect-exposing tests T_{bug}

$T_{keep}, T_{drop}, T_{bug} \leftarrow \emptyset$
for all $t \in T$ **do**
  $v \leftarrow GenValid(p)$
  $fails \leftarrow \sum_{i=1}^{k} [IsFail(ExecuteTest(t, v_i))]$
  if $fails = 0$ **then**
    $T_{keep} \leftarrow T_{keep} \cup \{t\}$
  else if $fails > k/2$ **then**
    $T_{drop} \leftarrow T_{drop} \cup \{t\}$
  else
    $T_{bug} \leftarrow T_{bug} \cup \{t\}$
  end if
end for
return $T_{keep}, T_{drop}, T_{bug}$

This process improve a common LLM limitation: the generation of overly restrictive or incorrect constraints for a parameter.

For GenValid(), we combine a constraint-based value injection with independent LLM. Injected valid values are generated by leveraging an LLM to synthesize candidate values based on parameter metadata (type, range, format constraints, and optional dependencies), followed by rule-based verification using the constraint-based checking from prior work [38, 72] to ensure syntactic and semantic valid. We do not claim novelty in generating and injecting parameter values, which have been explored by prior work on configuration constraint extraction and validation. Our contribution lies in leveraging such established value generation as a systematic means to evaluate the quality of generated configuration tests. We notice the theoretical limitation here: a bug that is extremely easy to trigger (e.g., a null pointer dereference on almost any valid input), or a bug that triggered only by rare but valid values (e.g., extreme values or specific combinations) could be misclassified. However, a truly subtle bug would usually fail only under specific values, not across a majority of valid inputs [37, 38].

Design Guideline for Challenge 2: A configuration test should not frequently fail within the spectrum of valid parameter values, otherwise it may assert the wrong constraints.

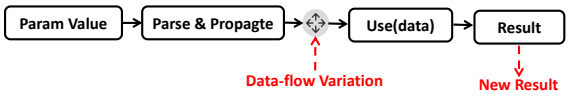
3.3.3 Conf-Sensitivity Verification. The first two steps ensure tests are syntactically correct and cover relevant code with valid assertions. However, a test can pass these stages yet still be semantically shallow. It may execute configuration-related code but fail to actually test the intended behavior controlled by the configuration parameter. For example, a test might only verify that a function logs something, rather than verifying that the log content is correctly truncated based on the specific parameter hadoop.caller.context.max.size. Although such tests do not produce false positives, they will incur maintenance and test-running overhead.

We introduces Conf-Sensitivity Verification (Algorithm 2). If the logic of how a configuration value influences program behavior is

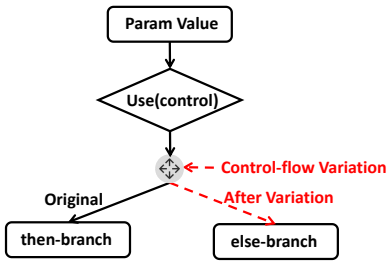
Algorithm 2 Configuration-Sensitivity Verification

Input: Candidate test t, Target parameter p
Output: Detection flag, Undetected variations

$type \leftarrow$ **IdentifyConfigType**(p)
$CR \leftarrow$ **LocateCRCode**(t, p)
$Ops \leftarrow$ **SelectVariations**$(CR, type)$
$r \leftarrow$ **ExecuteTest**(t)
for all $op \in Ops$ **do**
  $r' \leftarrow$ **ExecuteWithVariation**(t, op, CR)
  if $r' \neq r$ **then**
    return Detected
  end if
end for
return (**Undetected**, Ops)



(a) data flow variation



(b) control flow variation

Figure 5: Configuration semantic variation examples.

intentionally altered, a good test should produce a different result. Traditional mutation testing [14, 29, 32, 50, 82] cannot be directly applied because it can modify program semantics but cannot precisely modify configuration semantics. Therefore, we need to perform variations at the correct configuration code locations. We create a set of semantic variations for the identified configuration-related code. The catalog of semantic variation operators are shown in Table 1. These variations are logic-altering changes that mirror plausible changes in configuration semantics (e.g., flipping a comparison operator in a branch guarded by a configuration value). For each test, we generate 1-3 variants based on its variation strategy. Each candidate test is executed against both the original program and its semantically varied versions. A test is configuration sensitive only if it can detect these intentional semantic alterations—i.e., its execution outcome (pass/fail) changes.

We conduct targeted semantic variations guided by the data flow and control flow of configuration parameters. As shown in Fig. 5, we implement data-flow variations by altering the configuration value stored in the program variable to determine whether the test oracle can detect the corresponding changes in expected results. Similarly, we implement control-flow variations by modifying the execution path governed by a parameter, thereby assessing whether the test

Table 1: Configuration semantic variations used in our design, where p denotes the parameter value.

Operation	Original	Variation
① Arithmetic Operator Replacement	limit = p - v;	limit = p + v;
② Comparison Operand Modification	limit = min(p, c);	limit = min(p, 0);
③ Value Offset	threads = p;	threads = p - 1;
④ Value Scaling	timeoutMs = p;	timeoutMs = p * 1000;
⑤ Value Clamping	queueCap = p;	queueCap = max(c, p);
⑥ Condition Reversal	if (p > v) {A} else {B}	if (!(p > v)) {A} else {B}
⑦ Logical Connector Reversal	(p > v) && (v > c)	(p > v) || (v > c)
⑧ Loop Guard Reversal	while (p > c) {A}	while (p <= c) {A}

correctly asserts the behavioral outcome associated with that path. We conduct variations at the points where configurations interact with program variables/constants by locating the interaction points between configurations and program variables/constants (demonstrated in previous study [43]).

We give a case study in Fig. 7 (§ 4.4). We regard this process as an initial effort to bridge configuration semantics with mutation testing. Future work can design more comprehensive mutation strategies tailored specifically to configuration-related code.

Design Guideline for Challenge 3: An effective configuration test should be sensitive to semantic logic changes dictated precisely by the configuration.

3.4 Implementation

We implement approximately 7k lines of code in total. This includes enhancements of existing Java taint analysis tool (2k lines of Java code) and the implementation of a test generation and optimization pipeline (4k lines of Python code and 1k lines of Java code). We adhere to the design of ConfTainter [64] (a configuration taint analysis tool for C/C++) and improved the existing open-source Java taint analysis tool, Cflow [2] to extract configuration propagation paths for the identification of configuration-related (CR) code. During the test execution phase, we use Maven 3.6.3 [3] to run the tests and employ JaCoCo [1] to collect code coverage information. In our experiment, for each parameter, CTForge generates 2–15 test cases, with an average of 6 test cases per parameter.

4 Evaluation

We compare CTForge with LLM-based test generation and configuration validation tools, guided by the following research questions:

- **RQ1:** How effective is CTForge on detecting real-world configuration related issues? (§ 4.2)
- **RQ2:** What is the false positive rate of CTForge? (§ 4.3)
- **RQ3:** How does configuration-aware optimization improve test quality compared to preliminary LLM generated tests? (§ 4.4)
- **RQ4:** Can CTForge be practically used? (§ 4.5)

4.1 Dataset and Experiment Setup

To evaluate CTForge, we use representative large-scale software systems in configuration study and validation [39, 60] for experiments, which are HCommon, HDFS, HBase, Alluxio and ZooKeeper.

Table 2: Software systems used in our experiment.

Software	Description	#P	#Config Issues	#Injected Param/Value
HCommon	Hadoop engine	269	12	50/250
HDFS	Distributed file system	296	26	50/250
HBase	Distributed database	205	25	50/250
Alluxio	In-memory storage	515	4	50/250
ZooKeeper	Config management	32	12	32/160

Table 3: Effectiveness of different tools on real-world configuration-related issues.

Tool	Misconfig	Code Bug	Incompat. Issue	Total	Test Time
HITS	12/51 (23.5%)	2/13 (15.4%)	6/15 (40.0%)	20/79 (25.3%)	2–30 min
Ciri	45/51 (88.2%)	N/A	N/A	45/79 (57.0%)	20–60 s
CTest	41/51 (80.4%)	10/13 (76.9%)	2/15 (13.3%)	53/79 (67.1%)	20–230 min
CTForge	46/51 (90.2%)	9/13 (69.2%)	8/15 (53.3%)	63/79 (79.7%)	1–20 min

These systems are: 1) mature software from industry; 2) highly configurable; 3) with well-maintained documentation.

To validate the effectiveness of our tool in addressing real-world configuration-induced issues, we reuse the whole configuration issue dataset from existing work [60]. Specifically, we identify a special category of issues caused by software evolution: due to changes in CR code, values that were valid in previous versions now lead to bugs or functional inconsistencies in the updated code. Such issues were absent in existing datasets. Therefore, we obtain cases from the configuration evolution history dataset [78]. We filter out the cases that don't bring any incompatible issues (e.g., simply add new parameters or change default value). We identified and curated an additional 15 incompatible issues and incorporated them into our data set. Overall, the dataset contains 51 misconfigurations, 13 bugs, and 15 incompatible issues (Table 2).

We compare CTForge with state-of-the-art configuration validation tools: Ciri [39] and CTest [60]. Specifically, we also include the LLM-based test generation tool HITS [66] for comparison. Ciri leverages LLMs to check the value validity against the configuration parameter. CTest uses existing unit test to test configuration changes. HITS is an LLM-based general purpose unit test generation tool (instantiated with GPT-4o). We use HITS to synthesize unit tests with configuration-related code context.

In our experiments, GPT-4o-latest [27] (with the temperature set to 1.0) serves as the primary LLM engine for testing on real-world issues, aligning with common practice in the field [39, 66] where it is the top-performing model. To assess the generality of our approach, experiments on injected misconfigurations are conducted using both GPT-4o and Kimi-k2 [63], with parameters left at provider defaults. All experiments are conducted on Ubuntu 24.04 machines equipped with 128 GB of RAM and Java 1.8.0_442.

4.2 RQ1: Effectiveness of CTForge

As shown in Table 3, CTForge achieves the highest recall, detecting 63 out of 79 issues. However, the undetected cases (5 misconfigurations and 4 code bugs) highlight intrinsic limitations in both the knowledge encoded into the approach and the generative capacity of the underlying model. For misconfigurations, the failures are attributed to unmodeled configuration dependencies and

Table 4: New issues identified by CTForge.

Issue ID	Status	Issue Cause	Description
HBASE-28866	Confirmed & Fixed	Missing check	Setting `hbase.oldwals.cleaner.thread.size` to a negative value breaks HMaster and produces hard-to-diagnose logs.
HBASE-28881	Confirmed & Fixed	Missing check	Setting `hbase.master.procedure.threads` to negative value doesn't break HMaster but clients cannot connect
HDFS-17824	Confirmed & Fixed	Hidden constraint	DataNode fails to start when dfs.datanode.directoryscan.threads is set to 0.
HADOOP-19667	Confirmed	Hidden constraint	Clarifies legal value constraints for hadoop.security.crypto.buffer.size (minimum 512; rounded down to the cipher block size).
HADOOP-19665	Patch Submitted	Missing check	Setting hadoop.security.kms.client.encrypted.cache.expiry to a negative value causes `KMSClientProvider` initialization failure.
HADOOP-19673	Patch Submitted	Hidden constraint	Setting io.mapfile.bloom.error.rate to ≤ 0 or ≥ 1 causes NaN/zero vector size and writer construction failure.

the parser's default-value fallback when parameters are missing, revealing incomplete semantic knowledge about parameters. For bugs, CTForge's strategy primarily targets coverage of parameter usage sites, but it cannot fully capture and infer how it influences program behavior beyond those sites. As a result, bugs that lie in configuration-controlled branches and depend on specific functional scenarios, external inputs, or workload conditions are rarely triggered. For example, in `HADOOP-10508` [19], when RPC authorization (config) is enabled, the `RefreshCallQueueProtocol` interface was never registered, causing the NameNode to treat `refreshCallQueue` requests as unrecognized protocols. These difficulties are further amplified when parameters have complex value domains, inter and intra constraints, and subtle dependency structures. These are the cases that the current model cannot sufficiently reason about to generate effective tests. In contrast, CTest gains an edge in this context by executing all existing official unit tests that encode developers knowledge, thereby improving the capability of exposing such configuration-sensitive bugs.

Compared to other tools, CTForge is good at detecting software-evolution–induced inconsistencies in the use of configuration values (incompatible issues). This capability comes from its design: during test generation, CTForge explicitly employs semantic variations to selectively retain tests that effectively identify the variations. As a result, when developers modify configuration behaviors, these tests are highly sensitive to inconsistencies, enabling earlier detection of evolution-induced regressions. We may miss cases where brand new usages of existing parameters are introduced because tests are generated on the pre-change version and cannot exercise the new code paths; additionally, some inconsistencies are missed when test generation fails to instantiate the specific configuration values that trigger them.

HITS exhibits low effectiveness. Its LLM-synthesized tests often hard-code invalid configuration values and may encode the resulting exceptions as *expected* behavior, causing the tests to pass on misconfigurations and bugs. Ciri is capable of validating configuration values, but is also suffering from LLM hallucination, resulting in false alarms (will be discussed in RQ2). Also, simply check parameter value like Ciri is not capable of detecting code bugs or Incompatibility issues. The effectiveness of CTest is inherently constrained by the quality of the existing test suite. Moreover, because CTest depends on existing test suites and code changes must pass them before merging. Without intentionally altering values, CTest rarely reports evolution–induced inconsistencies.

Although CTForge is designed to generate tests for configuration validation purposes (such as integration or regression testing), its inherent capability to detect bugs through testing remains evident. During our experiments, it reveals 6 previously unknown issues (shown in Table 4). Our test cases can be integrated with fuzzing tools to perform fuzz testing, thereby enhancing their potential to discover unknown bugs.

4.3 RQ2: False Alarms of CTForge

Low false positive rate is a property that is essential for practical deployment where excessive warnings would undermine user trust. To evaluate the accuracy of our tool, we adopt the widely-used method of configuration value injection testing [38, 39, 60]. Specifically, we follow the approach of previous work by injecting values into various types of configuration parameters. This value generation process is independent of CTForge (e.g., generate valid value in § 3.3.2): it uses separate prompt templates, executes via independent API calls with no shared context or memory, and does not reuse any intermediate results from CTForge 's test generation or optimization pipeline. The generated value sets are created once per parameter and are used only as an evaluation oracle.

Since these software systems contain hundreds of configuration parameters, testing all parameters is not practical. Therefore, we adopt the random sampling methodology used in previous configuration-related studies and validation work [39, 43, 60]. For each system, we randomly select 50 configuration parameters (except for ZooKeeper-server, which only has 32 parameters, so all of them are included). For each parameter, we generate two valid values and three invalid values. In total, we collect 232 configuration parameters and inject 1160 values. We report precision, recall, and F1-score. To demonstrate that CTForge's effectiveness is not contingent on a specific LLM's capability, we employ two distinct model engines in our evaluation: GPT-4o and the Kimi-k2. Consistent performance (e.g., high F1-score in both models) confirms that the design of our framework is generalizable and not an artifact of the strengths of a single model.

Table 5 summarizes the results. The key strength of CTForge lies in its low false positive rate. This is achieved through a two stage: first, the CR code coverage enhancement ensures that configuration-handling logic is actually reached before any assertion is evaluated, filtering out tests that would otherwise produce spurious failures. Second, the assertion validity voting process exercises an admissible value set and requires oracles that generalize across it, which reduces wrong assertions.

This stands in marked contrast to alternative approaches that rely on LLM-generated tests as direct validators. For example, HITS

Table 5: Detection of injected misconfigurations.

Software	HITS			CTest			Ciri			CTForge-Kimi			CTForge-GPT4o		
	Recall	Precision	F1-score	Recall	Precision	F1-score	Recall	Precision	F1-score	Recall	Precision	F1-score	Recall	Precision	F1-score
HCommon	0.29	0.88	0.44	0.56	1.00	0.72	0.88	0.63	0.73	0.72	1.00	0.84	0.76	1.00	0.86
HDFS	0.37	0.86	0.52	0.78	1.00	0.88	0.82	0.58	0.68	0.74	1.00	0.85	0.86	1.00	0.92
HBase	0.23	0.83	0.36	0.60	1.00	0.75	0.80	0.64	0.71	0.75	1.00	0.86	0.70	1.00	0.82
Alluxio	0.21	0.86	0.34	0.54	1.00	0.70	0.82	0.57	0.67	0.67	1.00	0.80	0.72	1.00	0.84
ZooKeeper	0.24	0.88	0.38	0.67	1.00	0.80	0.84	0.60	0.70	0.81	1.00	0.90	0.93	1.00	0.96
Average	0.27	0.86	0.40	0.63	1.00	0.77	0.83	0.60	0.70	0.74	1.00	0.85	0.79	1.00	0.88

Table 6: CR-code coverage (%) w/ and w/o the coverage refinement: statement coverage (SC) and branch coverage (BC).

Software	CTForge-O0		CTForge-O1	
	SC	BC	SC	BC
HCommon	45.2	38.7	66.2	60.8
HDFS	50.1	42.3	71.4	65.7
HBase	40.6	40.2	68.9	59.5
Alluxio	44.9	37.5	62.5	57.8
ZooKeeper	48.3	41.0	60.7	50.9
Average	45.8	39.9	65.8	58.9

leverages generated unit tests to detect misconfigurations. The limited effectiveness of this strategy can be explained by three practical issues inherent to LLM-synthesized tests. First, these tests often achieve path reachability via hard-coded stimuli, yet their checks rarely provide semantic validation. Second, and most critically for false positives, some generated oracles overfit to default-oriented behavior (e.g., implicitly treating the default value as ground truth), which may misclassify non-default settings as failures, which would not arise in our voting-based validation.

CTest shows high precision but lower recall, reflecting the limited config-sensitivity of existing unit tests. Ciri achieves higher recall but lower precision, which reveals its vulnerability to false positives. This trade-off aligns with the observation that configuration testing cannot simply be replaced by LLM-based validation [39]. In contrast, CTForge's voting mechanism specifically targets the false-positive problem, making it more reliable for real-world configuration testing scenarios.

4.4 RQ3: Ablation Study of Test Optimization

Since the preceding steps follow a general LLM-based test generation pipeline (adapted here to use configuration-specific context), the ablation study is then designed specifically to check the impact of the three optimization stages, which constitute the main contributions of CTForge. By cumulatively enabling each optimization stage, the study demonstrates how each stage progressively improves test quality beyond naive LLM generation.

We include a baseline, CTForge-O0, which is the raw version that disables all optimization. Our ablation enables modules cumulatively: O1 (O0 + CR code coverage refinement), O2 (O1 + assertion validity voting) and O3 (O2 + configuration sensitivity verification, which is the final version of CTForge). We name these variants CTForge-O1, CTForge-O2, and CTForge-O3. All other settings are kept constant, and experiments are conducted on GPT4o.

Table 7: Effect of validity voting on the precision of detecting injected misconfigurations.

Variant	HCommon	HDFS	HBase	Alluxio	ZooKeeper	Avg.
CTForge-O1	0.97	0.98	0.95	0.96	0.95	0.96
CTForge-O2	1.00	1.00	1.00	1.00	1.00	1.00

Table 8: Effect of adding config-sensitive verification.

Tool	Misconfig	Code Bug	Incompatible Issue	Total
CTForge-O2	45/51 (88.2%)	4/13 (30.8%)	3/15 (20.0%)	52/79 (65.8%)
CTForge-O3	46/51 (90.2%)	9/13 (69.2%)	8/15 (53.3%)	63/79 (79.7%)

CR Code Coverage Refinement. Table 6 reports statement coverage (SC) and branch coverage (BC) of CR code with and without the coverage module. Across all five systems, enabling CTForge-O1 improves both the statement coverage and the branch coverage. On average, coverage improves by approximately 1.44× for SC and 1.48× for BC. These results confirm that coverage refinement is effective in generating tests that exercise a substantially larger fraction of configuration-related code, thereby strengthening the foundation for subsequent validation steps.

Assertion Validity Voting. Adding the assertion valid voting on top of coverage refinement improves precision by filtering out spurious cases where constraints are misinterpreted. As shown in Table 7, precision increases for all software. Based on our inspection, we find two dominant sources of false positives that generated by LLM: 1) misunderstanding of configuration constraints (Fig. 6a). The test asserts that `dfs.client.mmap.cache.size` must be greater than 0, overlooking the fact that zero-copy reads remain supported even when this size is configured to 0. 2) incorrect assertions of equality-to-default (Fig. 6b). The test incorrectly asserts equality between the configuration value and the default value. These two type of false alarms can be by eliminated.

Conf-Sensitivity Verification. As shown in Table 8, conf-sensitivity verification enhances the sensitivity of tests to configuration values and their interactions with CR code. Compared to CTForge-O2, CTForge-O3 increase the total detection rate from 65.8% to 79.7%. Specifically, the refinements are more effective at improving the ability to detect code bugs and incompatible issues. These results demonstrate that this stage is crucial for testing configuration-realted code logic, and thus improving overall effectiveness in real-world configuration-issue detection.

Fig. 7 presents an example that how this optimization works. The parameter `hadoop.caller.context.max.size` bounds the length

Valid Configuration:

```
dfs.client.mmap.cache.size = 0
```

```
@Test
public void testDfsClientMmapCacheSizeConfigValidity() {
    //Read the configuration value
    int mmapCacheSize = conf.getInt(···, ···);
    //The value of dfs.client.mmap.cache.size must be > 0
    Assert.assertTrue(···, mmapCacheSize > 0);
}
```

(a) An FP case caused by incorrect constraint synthesis.

Valid Configuration:

```
hbase.mob.cache.evict.period = 1800
```

```
@Test
public void testMobCacheEvictPeriodDefaultValue() {
    //Retrieve the configuration value
    long mobCacheEvictPeriod = configuration.getLong(···, ···);
    //Validate equality between configuration and default value
    assertEquals(3600, mobCacheEvictPeriod);
}
```

(b) An FP case caused by incorrect assertions.

Figure 6: False positive (FP) examples when running configuration tests.

```
hadoop.caller.context.max.size = 128
callerContext.getContext().length() = 256
```

```
public void logAuditEvent(···) {
   if (callerContext.getContext().length() >
callerContextMaxLen) {
     sb.append(···.substring(0,callerContextMaxLen));
     } else {
          sb.append(callerContext.getContext());
        }
    logAuditMessage(sb);  //write to lastLog
}
/*FSNamesystem.java */
```

```
@Test
public void loggerTest()  {
    ···
    TestAuditLogger L= new TestAuditLogger();
    L.logAuditEvent(··· );
    if(callerContext.getContext().length() >
        hadoop.caller.context.max.size){
    assertNotNull(L.lastLog); //ineffective assert
+   assertTrue(L.lastLog.length() == MaxLen);
    }
}
```

(a) Test result on original code.

```
hadoop.caller.context.max.size = 128
callerContext.getContext().length() = 256
```

```
public void logAuditEvent(···) {
   if (callerContext.getContext().length() <=
callerContextMaxLen) {
     sb.append(···.substring(0,callerContextMaxLen));
     } else {
          sb.append(callerContext.getContext());
        }
    logAuditMessage(sb);  //write to lastLog
}
/*FSNamesystem.java */
```

```
@Test
public void loggerTest()  {
    ···
    TestAuditLogger L= new TestAuditLogger();
    L.logAuditEvent(··· );
    if(callerContext.getContext().length() >
        hadoop.caller.context.max.size){
    assertNotNull(L.lastLog); //ineffective assert
+   assertTrue(L.lastLog.length() == MaxLen);
    }
}
```

(b) Test result after configuration semantic variation.

Figure 7: An example of how configuration semantic variation can distinguish effective test assertions.

of the caller context serialized into audit logs. In `logAuditEvent`, this bound guards a truncation branch: if the context length exceeds the configured limit, the string is truncated to `callerContextMaxLen` before logging, otherwise it is logged intact. A preliminary test asserts `assertNotNull(log)`. This test passes regardless of whether the truncation logic works. During configuration-sensitivity verification, we create a variation where the truncation condition is reversed. The preliminary test still passes on both versions, revealing its ineffectiveness. In contrast, a refined test that asserts the exact length of the logged message after truncation would pass on the original logic but fail on the varied logic, proving its semantic sensitivity to this configuration parameter.

4.5 RQ4: Use CTForge in Practice

Our evaluation is conducted on real-world systems rather than small code snippets, using their standard build and test infrastructure (Maven), demonstrating CTForge's utility in industrial-level software. However, there are still two factors may limit the practical usage of CTForge, which we further discuss here.

Consistency of Test Quality. We notice that LLM-based generation is inherently non-deterministic. CTForge inherits such stochasticity and can not generate identical test code across multiple runs. CTForge mitigates such non-determinism by providing the CR code corresponding to the parameter to prevent the LLM from generating random tests or relying solely on its training knowledge, and further employs optimizations to enhance the test's correctness and relevance. To investigate whether the tool can consistently produce effective results, we independently conducted another two additional replicate experiments, and then compared the results across the three experiments. We identify CTForge's fluctuation on misconfiguration recall and CR-code coverage.

- **Real-world issue detection.** For the detection of real-world configuration issues, the three experiments identified 60, 62, and 63 issues respectively. Among these, the numbers of detected Code Bugs and Incompatible Issues remained consistent, whereas the counts for Misconfigurations differed (43,45,46). This variation is attributed to the diverse types and forms of misconfigurations; the undetected misconfigurations were those for which the generated specific value validity tests (Fig. 4(a)) were not comprehensively or un-runnable(e.g., missing checking a value dependency with other variables).
- **False positive rate.** As shown in Fig. 8, CTForge consistently achieves high precision, which is due to the Validity Voting

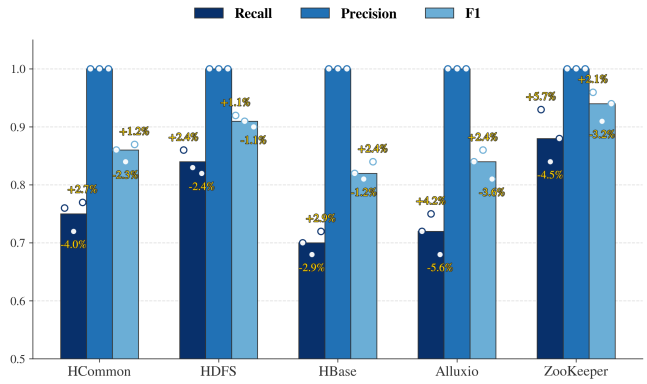


Figure 8: Repeated results of injected misconf detection. The scattered points represent the results from each single run.

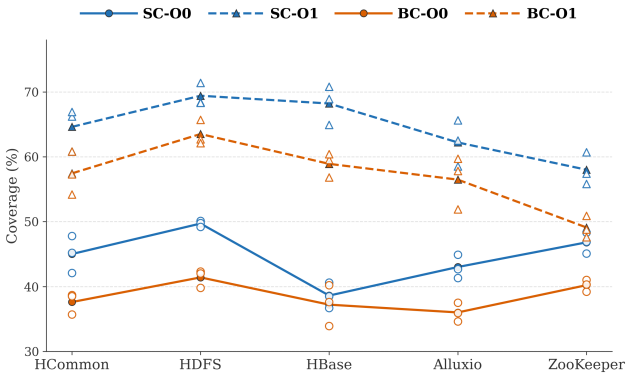


Figure 9: Repeated results of CR-code coverage. The scattered points represent the results from each single run.

process. However, the recall (i.e., the detection rate for misconf) exhibits slight variations, with its fluctuation range spanning from −5.6% to +5.7%.

- **CR-code coverage.** As shown in Fig. 9, there are some differences in CR-code coverage, because CTForge cannot guarantee generating identical code. Note that the BS and SC of O1 are consistently higher than those of O0, demonstrating the effectiveness of the Code Coverage Refinement.

The results reflects one limitation of using CTForge in practice: Despite the strong generative capabilities of LLM, it cannot guarantee consistent high coverage of the entire configuration space and generate comprehensive assertions in every run. That is to say, we strive to leverage the configuration semantics to ensure the correctness and validity of the test generation. But in each run, we cannot ensure that the model's maximum capability is fully exploited to generate thorough tests. Future work could: 1) take into account the possible coverage gains achieved through multiple re-runs and the associated time trade-offs; 2) employ more powerful program analysis tools to provide more precise CR code snippets.

Test Time Overhead. Testing, particularly for large scale systems, can be time-consuming, and generation requires non-negligible upfront resources. We report the test execution time (1–20 min per parameter as shown in Table 5), following previous work experiment [12, 60]). This time directly impacts developers' workflow as the time will be accumulated over repeated CI/CD runs. The end-to-end generation cost of CTForge for one test completes within 181–1173 seconds: preliminary test generation takes 42–136 seconds, CR code coverage enhancement takes 15–364 seconds, assertion validity voting takes 37–340 seconds, and configuration-sensitivity verification takes 46–605 seconds (The test with the longest total time is not necessarily the one in each individual stage). Once generated, the test suite can be committed to the repository and reused across versions as regression test, amortizing the upfront generation cost over time.

Although with such overhead, CTForge is usable. Users can leverage these tests into their integration or regression testing processes. These tests are designed at the granularity of parameter level, allowing developers or users to selectively generate/run tests targeting specific parameters during production use. For example, when a developer changes the usage of one parameter, he can then run its corresponding test without running the whole regression test set. Furthermore, CTForge can integrate with established test prioritization and selection techniques [10, 12, 35, 42, 84]. Although the current implementation does not explicitly include these optimizations, they represent promising extensions for future work that enables a more scalable and efficient approach to configuration testing in large-scale deployment scenarios.

5 Threats to Validity

Internal Validity. Our testing is incomplete. Unlike formal verification that can provide rigorous verification, our approach cannot certify the absence of configuration bugs in the code that passes our tests. Instead, it aims to reduce the likelihood of such bugs across the majority of common configuration scenarios.

The effectiveness of CTForge is highly dependent on the quality of the test oracles generated by the LLM. The inherent uncertainty problem of generative model remains a challenge. First, LLMs may generate insufficient test oracles that fail to cover the entire space of possible errors for configuration parameters. Second, LLMs may generate semantically incorrect assertions, such as asserting overly restrictive constraints. Although our validity voting (§ 3.3.2) mitigates this issue by executing tests against multiple valid parameter values, for parameters with large or continuous value spaces (e.g., path-type parameters), exhaustive enumeration is infeasible. Human expertise remains indispensable for precisely delimiting the valid ranges (e.g., through documentation).

Our approach relies on taint analysis to identify CR Code. The CR Code provided to LLMs directly effect the coverage of generated tests. However, static analysis is imprecise and hard to scale, especially in large-scale systems with complex data and control flows. This imprecision may lead to: 1) incomplete identification of CR Code, resulting in tests that fail to cover critical configuration-code interactions, and 2) false positives in CR Code identification, causing the generation of irrelevant tests that do not actually exercise configuration. We mitigate this threat by iteratively refining test coverage (§ 3.3.1) and filter out irrelevant tests by configuration sensitivity verification (§ 3.3.3).

External Validity. We choose Java-based cloud systems in our study, as existing research on runtime configuration validation has

predominantly focused on them. Although we argue that the core concepts of our approach are general, the performance may be various in other languages. First, the precision of taint analysis is sensitive to the underlying language's type system and alias analysis capabilities. Second, the control-flow complexity may varies across languages. Consequently, even with identical designs, our method's bug-detection rate and runtime overhead may differ. The implementation to other languages would require engineering effort to adapt: 1) re-implement taint analysis using language-specific frameworks, 2) the configuration parsing and loading mechanisms vary across languages, requiring customized prompts, 3) the generated test must adapt to divergent test-runner semantics.

Construct Validity. The primary threats to construct validity concern the representativeness and provenance of our experimental dataset (§ 4.1). To mitigate these risks, we deliberately construct our benchmark from well-established, peer-reviewed published datasets, which reflects common practice. As a result, our evaluation may inherits any underlying selection biases or incompleteness present in that legacy dataset. We mitigate this threat by integrating two datasets, thereby improve the complementarity of the data.

6 Related Work

Software Production Line Configuration. This paper specifically focuses on runtime configuration validation. While configuration validation on the software production line is also essential and shares some underlying principles [13, 33, 47–49, 69], they differ in scope and focus [46, 78]. Specifically, feature toggle is a form of runtime configuration, and we see our work as complementary to, rather than overlapping with that established literature. We explicitly follow the distinction advocated by [46]. Configuration parameter are permanent, user-facing, and often have complex constraints and dependencies, which requires systematic testing across a large configuration space. Feature toggles are temporary, developer-controlled, used for continuous deployment, A/B testing, or hiding incomplete features, and are typically tested on a few values (on/off). Some of the techniques in feature toggle configuration testing [4] can be applied to our testing pipeline. For examples, variability-aware static analysis [53], option sampling [31, 45, 70] and test prioritization [23, 41].

We are positive that several core techniques in CTForge can be reused or adapted to test feature toggles. CR-code coverage enhancement can be repurposed to ensure tests exercise the code guarded by a toggle, regardless of whether the toggle is temporary. Configuration-sensitivity verification is also usable as the semantic variations for toggles are more straightforward. However, some parts of CTForge are less relevant: the two-tier value-validity testing (since toggles typically have no rich value spaces). Also, CTForge generates regression tests, while toggles are ideally short-lived, so the test suite may need a different lifecycle management.

Configuration Testing. Some existing configuration tests target at fuzzing configuration values to find bugs, including grammar-based [34, 65, 81] and constraint-based fuzzing [37, 38]. By leveraging program analysis or machine learning, these approaches generate parameter values that violate certain constraints or generate valid but boundary/extreme values to expose latent bugs. Another category of fuzzing work integrates configuration as an additional input dimension into existing fuzzing pipelines [21, 57, 61], and has achieved promising results on complex system software such as the Linux kernel. These works are different from our work as they build on existing test frameworks or test suites, while we try to generate test suites as basic test infrastructure. The tests generated by CTForge can be integrated into these fuzzing frameworks.

CTest [60] connects configuration values with existing developer-written software tests. Although effective, CTest relies on existing test suites and lacks systematic methods for automatically generating tests that specifically target configuration parameters, limiting their scalability and coverage of a large number of parameters.

Constraint-based Configuration Checking. It commonly relies on extracting configuration constraints to verify configuration values. Manually developing validators requires considerable domain expertise and is time-consuming [6, 26, 51, 62]. To reduce this cost, ML/NLP-based configuration validation techniques have been explored. Traditional ML/NLP approaches learn correctness rules from configuration data [8, 77] and documentation [68], and then apply these learned rules for validation. More recently, LLMs have been used for configuration validation [39].

A fundamental limitation of these methods is that they only perform static validation by checking parameter values against a set of inferred or predefined rules. This is inherently limited because it cannot dynamically exercise these values within the actual code base, which is the advantage of configuration-specific testing.

Generating Tests by LLMs. Recent advances in large language models (LLMs) have shown strong potential for code and test generation [15–17, 44, 67, 80], with works such as ChatTester [74], TestGenEval [28], and Python-focused studies [9] advancing evaluation and methodology. Various tools have emerged, including TestPilot [56], TestGen-LLM [5], CODAMOSA [36], SymPrompt [54], and specialized efforts for JSON libraries [83]. Complementary research targets test quality through guided oracles [24], disoised training data [75], refactoring [18] and mocking analysis [85]. These work focus on general purpose unit test generation evaluated on standardized benchmarks, which may not practical on large-scale systems.

7 Conclusion

We present CTForge, an automated framework that uses LLMs to generate configuration tests to prevent configuration-induced failures in large-scale software. By integrating taint analysis with LLM-based synthesis, CTForge identifies configuration related code and trace value propagation across the system. Its core contribution is a configuration-aware optimization pipeline that iteratively refines raw LLM outputs—enhancing coverage, validating assertions, and verifying semantic sensitivity—thereby effectively mitigating hallucinations and improving test precision. Evaluated on five widely-used systems, CTForge outperforms existing configuration validation and testing work.The results demonstrate that structured, semantics-guided refinement is essential to produce reliable and effective LLMs configuration test suites, advancing the state-of-the-art in automated configuration validation.

8 Data Availability

The code and dataset can be found in the repository: https://zenodo.org/records/21359932.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This research was funded by NSFC (No.62532008, No.U2441238, No.62502528, No.62272473) and National University of Defense Technology Research Project (No.26-ZZCX-JDZ-18, No.ZK24-01).

References

[1] 2023. JaCoCo: library for measuring and reporting code coverage in Java applications. https://www.jacoco.org/jacoco/.

[2] 2024. cFlow: Flow-based Configuration Analysis Framework for Java Bytecode Based Cloud Systems. https://github.com/xlab-uiuc/cflow

[3] 2024. Maven: a build tool for Java projects. https://maven.apache.org/.

[4] Halimeh Agh, Aidin Azamnouri, and Stefan Wagner. 2024. Software product line testing: a systematic literature review. *Empirical Software Engineering* 29, 6 (2024), 146.

[5] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 185–196.

[6] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*. 29–35.

[7] Farnaz Behrang, Myra B Cohen, and Alessandro Orso. 2015. Users beware: Preference inconsistencies ahead. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 295–306.

[8] Ranjita Bhagwan, Sonu Mehta, Arjun Radhakrishna, and Sahil Garg. 2021. Learning patterns in configuration. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 817–828.

[9] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit test generation using generative AI: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 54–61.

[10] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667.

[11] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 362–374.

[12] 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*. 452–465.

[13] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 129–139.

[14] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.

[15] Chunhao Dong, Yanjie Jiang, Yuxia Zhang, Yang Zhang, and Liu Hui. 2025. ChatGPT-Based Test Generation for Refactoring Engines Enhanced by Feature Analysis on Examples. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 746–746.

[16] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–38.

[17] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering* (2024).

[18] Yi Gao, Xing Hu, Xiaohu Yang, and Xin Xia. 2025. Automated Unit Test Refactoring. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 713–733.

[19] HADOOP-10508. 2014. RefreshCallQueue fails when authorization is enabled. https://issues.apache.org/jira/browse/HADOOP-10508.

[20] Navid Bin Hasan, Md Ashraful Islam, Junaed Younus Khan, Sanjida Senjik, and Anindya Iqbal. 2025. Automatic High-Level Test Case Generation using Large Language Models. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 674–685.

[21] Sanan Hasanov, Stefan Nagy, and Paul Gazzillo. 2025. A little goes a long way: Tuning configuration selection for continuous kernel fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 795–807.

[22] HBASE-15439. 2017. getMaximumAllowedTimeBetweenRuns in ScheduledChore ignores the TimeUnit. https://issues.apache.org/jira/browse/HBASE-15439.

[23] Robert M Hierons, Miqing Li, Xiaohui Liu, Jose Antonio Parejo, Sergio Segura, and Xin Yao. 2020. Many-objective test suite generation for software product lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–46.

[24] Soneya Binta Hossain, Raygan Taylor, and Matthew Dwyer. 2025. Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1870–1891.

[25] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.

[26] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. Confvalley: A systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[27] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[28] Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2024. Testgeneval: A real world unit test generation and test completion benchmark. *arXiv preprint arXiv:2410.00752* (2024).

[29] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[30] Dongpu Jin, Myra B Cohen, Xiao Qu, and Brian Robinson. 2014. Preffinder: Getting the right preference in configurable software systems. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 151–162.

[31] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1084–1094.

[32] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering* 23, 4 (2018), 2426–2463.

[33] Elias Kuiter. 2025. Towards Effective and Efficient Feature-Model Analyses for Evolving System Software. In *Proceedings of the 29th ACM International Systems and Software Product Line Conference-Volume B*. 6–13.

[34] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 220–231.

[35] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 583–594.

[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[37] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. Ecfuzz: Effective configuration fuzzing for large-scale systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[38] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and opportunities: an in-depth empirical study on configuration error injection testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 478–490.

[39] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. 2024. Large language models as configuration validators. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 204–216.

[40] Xiangke Liao, Shulin Zhou, Shanshan Li, Zhouyang Jia, Xiaodong Liu, and Haochen He. 2018. Do you really know how to configure your software? configuration constraints in source code may help. *IEEE Transactions on Reliability* 67, 3 (2018), 832–846.

[41] Jackson A Prado Lima, Willian DF Mendonça, Silvia R Vergilio, and Wesley KG Assunção. 2020. Learning-based prioritization of test cases in continuous integration of highly-configurable software. In *Proceedings of the 24th ACM conference on systems and software product line: Volume A-Volume A*. 1–11.

[42] Yu Liu, Jiyang Zhang, Pengyu Nie, Milos Gligoric, and Owolabi Legunsen. 2023. More precise regression test selection via reasoning about semantics-modifying changes. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 664–676.

[43] Chaopeng Luo, Yuanliang Zhang, Haochen He, Zhouyang Jia, Teng Wang, Shulin Zhou, Si Zheng, and Shanshan Li. 2024. Who is in Charge here? Understanding

How Runtime Configuration Affects Software along with Variables&Constants. In *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 363–372.

[44] YINGWEI MA, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. [n. d.]. At Which Training Stage Does Code Data Help LLMs Reasoning?. In *The Twelfth International Conference on Learning Representations*.

[45] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*. 643–654.

[46] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring differences and commonalities between feature flags and configuration options. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 233–242.

[47] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analysing the impact of workloads on modeling the performance of configurable software systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2085–2097.

[48] 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 *Proceedings of the 33rd acm/ieee international conference on automated software engineering*. 155–166.

[49] Damir Nešić, Jacob Krüger, Ștefan Stănciulescu, and Thorsten Berger. 2019. Principles of feature modeling. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 62–73.

[50] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 910–921.

[51] Markus Raab and Gergö Barany. 2017. Challenges in validating FLOSS configuration. In *IFIP International Conference on Open Source Systems*. Springer, 101–114.

[52] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*. 131–140.

[53] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-aware static analysis at scale: An empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4 (2018), 1–33.

[54] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971.

[55] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software configuration engineering in practice interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering* 46, 6 (2018), 646–673.

[56] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[57] Yuheng Shen, Jianzhong Liu, Yuhan Chen, Yifei Chu, Qiang Zhang, Guoyu Yin, Heyuan Shi, and Yu Jiang. 2026. Configuration-Sensitive Linux Kernel Fuzzing. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. Rio de Janeiro, Brazil, 1–12.

[58] Jonathan Shieber. 2019. Facebook blames a server configuration change for yesterday's outage.

[59] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering*. 313–322.

[60] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 735–751.

[61] Yue Sun, Yan Kang, Chenggang Wu, Kangjie Lu, Jiming Wang, Xingwei Li, Yuhao Hu, Jikai Ren, Yuanming Lai, Mengyao Xie, et al. 2025. SyzParam: Incorporating Runtime Parameters into Kernel Driver Fuzzing. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 1484–1498.

[62] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at facebook. In *Proceedings of the 25th symposium on operating systems principles*. 328–343.

[63] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534* (2025).

[64] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. 2023. Conftainter: Static taint analysis for configuration options. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1640–1651.

[65] Zi Wang, Ben Liblit, and Thomas Reps. 2020. Tofu: Target-oriented fuzzer. *arXiv preprint arXiv:2004.14375* (2020).

[66] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.

[67] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328.

[68] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. 2020. {PracExtractor}: Extracting configuration good practices from manuals to detect server misconfigurations. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 265–280.

[69] Yi Xiang, Han Huang, Sizhe Li, Miqing Li, Chuan Luo, and Xiaowei Yang. 2023. Automated test suite generation for software product lines based on quality-diversity optimization. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–52.

[70] Yi Xiang, Han Huang, Yuren Zhou, Sizhe Li, Chuan Luo, Qingwei Lin, Miqing Li, and Xiaowei Yang. 2022. Search-based diverse sampling from real-world software product lines. In *Proceedings of the 44th International Conference on Software Engineering*. 1945–1957.

[71] Tianyin Xu and Owolabi Legunsen. 2019. Configuration Testing: Testing Configuration Values as Code and with Code. *arXiv preprint arXiv:1905.12195* (2019).

[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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[73] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 159–172.

[74] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[75] Junwei Zhang, Xing Hu, Shan Gao, Xin Xia, David Lo, and Shanping Li. 2025. Less Is More: On the Importance of Data Quality for Unit Test Generation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1293–1316.

[76] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, 1–30.

[77] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 687–700.

[78] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 188–200.

[79] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2025. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *Computational Linguistics* (2025), 1–46.

[80] Yuanliang Zhang, Yifan Xie, Shanshan Li, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng Luo, Zhizheng Zheng, et al. 2025. Unseen Horizons: Unveiling the Real Capability of LLM Code Generation Beyond the Familiar. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 619–619.

[81] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing configurations of program options. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–21.

[82] Yifan Zhao, Yizhou Chen, Zeyu Sun, Qingyuan Liang, Guoqing Wang, and Dan Hao. 2024. Spotting Code Mutation for Predictive Mutation Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1133–1145.

[83] Zhiyuan Zhong, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. 2024. Advancing bug detection in Fastjson2 with large language models driven unit test generation. *arXiv preprint arXiv:2410.09414* (2024).

[84] Jianyi Zhou, Junjie Chen, and Dan Hao. 2021. Parallel test prioritization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–50.

[85] Hengcheng Zhu, Valerio Terragni, Lili Wei, Shing-Chi Cheung, Jiarong Wu, and Yepang Liu. 2025. Understanding and Characterizing Mock Assertions in Unit Tests. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 554–575.