FlyCatcher: Neural Inference of Runtime Checkers from Tests

BEATRIZ SOUZA*, University of Stuttgart, Germany
CHANG LOU, University of Virginia, USA
SUMAN NATH, Microsoft Research, USA
MICHAEL PRADEL, CISPA Helmholtz Center for Information Security, Germany

Complex software systems often suffer from silent failures, i.e., violations of the intended semantics that do not cause explicit errors. A promising approach to detect such errors is to use system-specific runtime checkers that monitor the execution of a system and check for violations of the intended semantics. However, writing such checkers for a given software system is challenging and time-consuming, and hence, rarely done in practice. This work presents FlyCatcher, an automated approach to derive runtime checkers from existing tests, i.e., from a resource available for most software systems. The critical challenge of such an approach is to generalize the behavioral properties encoded in a test case to arbitrary executions of a system. FlyCatcher addresses this challenge through a combination of LLM-based synthesis, static analysis, and dynamic validation, which infers a checker that monitors specific method calls and asserts properties that should hold when they are called. The inferred checkers are stateful, i.e., they reason about the system's behavior by maintaining a shadow state that abstracts the actual system state as needed by the checker. Our evaluation applies FlyCatcher to 400 tests from four widely used, complex software systems. The approach infers 334 checkers, out of which 300 are found to be correct via cross-validation. Compared with a state-of-the-art approach, our approach infers 2.6x more correct checkers, which enables it to detect 5.2x more errors. By contributing to the automated inference of runtime checkers from tests, this work enables the broader adoption of runtime checking as a practical approach to detect silent failures in complex software systems.

1 INTRODUCTION

Complex software systems suffer from bugs that elude traditional testing and formal verification. In particular, software systems often experience *silent failures*, i.e., violations of the expected semantics that do not surface as explicit errors. Despite the lack of overt symptoms, silent failures can corrupt data, produce incorrect results, and introduce security vulnerabilities. Recent studies report that such failures are prevalent in production deployments, e.g., accounting for 39% of failures examined by Lou et al. [42]. These failures are notoriously hard to debug as their effects may be detected only much later, long after the root cause has vanished. Consequently, there is a strong need for techniques that quickly detect silent failures as they occur.

A promising approach to detect such bugs is to deploy *semantic checkers* that run alongside the software and continuously verify that runtime behavior conforms to the intended semantics. The properties that such checkers should verify are highly domain-specific, e.g., in-order message delivery, correct session timeout handling, or maintaining replication invariants in a distributed system. Hence, manually writing semantic checkers typically demands deep domain expertise and painstaking manual effort. The scale, complexity, and semantic richness of real-world software systems make manual checker construction both challenging and brittle.

Prior work has explored techniques for automatically inferring semantic checkers. For example, the pioneering Daikon system [24] mines invariants statistically from execution traces. While useful, such invariants often capture relationships among low-level events rather than high-level semantics that are meaningful to developers. To address this gap, T2C [44] proposes deriving semantic checkers from existing tests, i.e., a resource that is anyway available for many software

*Work done partly as an intern at Microsoft Research.

Authors' addresses: Beatriz Souza, beatrizbzsouza@gmail.com, University of Stuttgart, Germany; Chang Lou, University of Virginia, USA, lchlou@virginia.edu; Suman Nath, Microsoft Research, USA, Suman.Nath@microsoft.com; Michael Pradel, CISPA Helmholtz Center for Information Security, Germany, michael@binaervarianz.de.

projects. This kind of approach leverages the insight that the carefully designed workloads and assertions encoded in tests convey rich knowledge of the expected system behavior. The knowledge captured in tests reflects the developers' intent: tests concretely demonstrate what the system should do and why. Intuitively, if we can lift these concrete, workload-specific tests into general, runtime checkers, we obtain checkers that are both semantically grounded and operationally useful.

Automatically converting tests into semantic checkers involves multiple challenges, though. First, tests and their assertions are crafted for specific workloads, whereas runtime checkers must operate on arbitrary executions of the system. For example, consider a test that creates an ephemeral data structure in a distributed system, assigns a specific name to it, and asserts that the data structure is removed after a timeout. This test cannot be used verbatim as a semantic checker because other executions of the system may create arbitrarily named data structures. T2C [44] uses static analysis to generalize tests into parameterized checkers (e.g., over the data structure name). However, this generalization is challenging because tests may contain multiple concrete values that must be parameterized and often intermix several assertions whose logic is intertwined with test scaffolding. The challenge is to disentangle these elements and generalize them correctly.

Second, generating effective checkers requires reasoning about constants and their meaning. In particular, tests commonly include magic values and domain-specific constants. Prior work [44] can fail to account for the role these values play, producing incorrect generalizations. For example, consider this test: `list = new List(); list.add(1); assert(list.length == 1);`

A purely structural analysis is unable to distinguish whether the value 1 in the assertion is (a) the value added, (b) a constant, or (c) a value tied to internal object state. As a result, a generated checker might incorrectly tie `length` to the value added, e.g., asserting `length == 5` after adding element 5, rather than to the count of elements.

Finally, many meaningful properties are stateful. For example, validating a list's length, the number of active sessions, or the membership of a replica set requires tracking how operations update system state over time. Designing stateful checkers requires (a) identifying which operations affect state, (b) reasoning about how system operations update state, and (c) maintaining complete coverage of all state-affecting operations. For example, a property that represents the length of a list must be incremented on `add()` and decremented on `delete()` operations. Without completeness, the state expected by the checker will drift from the true state, producing false positives or missed failures. Existing approaches do not support such comprehensive stateful checking.

This paper presents FlyCatcher, a novel approach for automatically inferring semantic checkers from existing tests by combining large language model (LLM)-based synthesis, static analysis, and dynamic validation. Given a test case as input, the approach generates a semantic checker that captures the high-level intent of the test and can be deployed in production to detect silent failures. The approach combines lightweight static analysis, e.g., to identify the methods called in the test, with LLM prompting, e.g., to generate candidate checkers. To increase the correctness of generated checkers, FlyCatcher employs a feedback loop that validates candidates via static analysis and by executing them on a suite of validation tests. When discrepancies arise, FlyCatcher provides structured feedback to the LLM to refine the checker iteratively.

Unlike prior work, our approach can generate checkers that are substantially more general, robust to diverse workloads, and capable of reasoning about magic values and domain-specific constants. Our intuition is that LLMs can leverage semantic context – including code, comments, and identifier names – to infer why a test asserts what it asserts, rather than merely what it asserts. The LLM uses this context to synthesize checkers that target high-level semantics rather than low-level event correlations. This ability enables correct parameterization (e.g., distinguishing element values from the element count of a list) and better generalization beyond a test's original workload. Another contribution of FlyCatcher is its ability to generate stateful checkers. Each

inferred checker maintains a checker-specific abstraction of the system's state, called the *shadow state*, and uses it to validate stateful properties. At runtime, the checker continuously compares the observed system state against the shadow state and reports any discrepancies. Conceptually, a checker that maintains a semantically faithful shadow state functions like a lightweight reference model: Whenever the implementation deviates from this reference, the mismatch signals a semantic failure.

We implement FlyCatcher for Java and evaluate it on 400 test cases from four complex open-source systems. Given these tests, the approach generates 334 semantic checkers, showing its ability to support a wide range of testing scenarios. Cross-validating these checkers against held-out tests, we find 300 of them to be correct, which increases the number of correctly inferred checkers by 2.6x over the state-of-the-art T2C approach [44]. To evaluate the effectiveness of the generated checkers at detecting silent failures, we use mutation testing and finds the FlyCatcher-generated checkers to detect 5.2x more mutants than T2C-generated checkers. The costs of using our approach are modest: On average, generating a checker takes 15 seconds and 183k LLM tokens (USD 0.60) and running a checker imposes a runtime overhead up to 2.7%–40.3% (depending on the target system).

In summary, this paper makes the following contributions:

- We present the first LLM-based approach to infer semantic runtime checkers from tests, which results in substantially more general and robust checkers than when using an existing static-analysis-based technique.
- We introduce a shadow state mechanism that allows the inferred checkers to maintain a checker-specific abstraction of the system's state and reason about stateful properties.
- We implement our ideas in FlyCatcher,[1] evaluate it on four real-world Java systems, and find that it significantly outperforms the best available approach in terms of correctly inferred checkers and bug detection ability.

2 MOTIVATING EXAMPLE AND PROBLEM DEFINITION

The following illustrates the kind of programming error we are addressing with an example based on Zookeeper, a widely used distributed system project, and then defines the problem addressed by our work. As a motivating example, consider a hypothetical bug in the `getChildren` method of the `DataNode` class of Apache Zookeeper, as shown in Figure 1. The method is supposed to return a set of children nodes. However, due to a programming error, it always returns an empty set, regardless of the actual state of the object. This bug can lead to silent failures where the system behaves incorrectly without crashing or throwing exceptions.

As testing is limited to a specific set of inputs and scenarios, such bugs can easily be missed. Indeed, the bug in Figure 1 remains undetected when running the comprehensive test suite of Zookeeper. Interestingly, there are existing test cases that check the correctness of the `getChildren` method, such as the test shown in Figure 2. The existing test case performs the following steps: 1) Instantiate a `DataNode` object and call `getChildren`; 2) Assert that the returned set is non-null and empty; 3) Add and remove a child, then repeat the assertions. These assertions capture important semantic properties, namely the correctness of `getChildren` under different states. However, the test is limited to its specific workload and assertions, and thus cannot catch the bug in Figure 1.

To generalize the bug detection capability of such tests, we aim to transform them into runtime checkers that monitor and assert the semantic properties encoded in tests, but generalized to work on arbitrary executions of the system. We call this problem the *test-to-checker* problem, defined as follows: Given a test t and two sets of additional tests, context tests $T_{context}$ and validation tests T_{val} with $t \notin T_{context} \cup T_{val}$, our goal is to generalize t into a runtime checker c. The runtime checker

[1] Our implementation is available at https://github.com/biabs1/FlyCatcher.

```
public synchronized Set<String> getChildren() {
    if (children == null) {
        return EMPTY_SET;
    }
    return Collections.emptySet();
}
```

Fig. 1. Example of bug missed in Zookeeper. The highlighted line is the faulty line that causes the method to always return an empty set.

```
public void
      testGetChildrenShouldReturnEmptySetWhenThereAreNoChidren
      () {
    // create DataNode and call getChildren
    DataNode dataNode = new DataNode();
    Set<String> children = dataNode.getChildren();
    assertNotNull(children);
    assertEquals(0, children.size());

    // add child,remove child and then call getChildren
    String child = "child";
    dataNode.addChild(child);
    dataNode.removeChild(child);
    children = dataNode.getChildren();
    assertNotNull(children);
    assertEquals(0, children.size());
}
```

Fig. 2. Example of test case in Zookeeper.

monitors invocations of state-changing methods exercised by t and validates that the state of the system satisfies the semantic properties encoded in t. The context tests in $T_{context}$ are used by our approach to provide additional semantic information about the system under test, which helps the generalization of t into c. The validation tests in T_{val} are used by our approach to ensure that the generated checker c is not overfitted to t and can generalize to other scenarios.

Addressing this problem involves three key challenges not sufficiently addressed by prior work:

(1) *Generalizing workloads.* The test t encodes a specific workload, but the checker c should be able to handle arbitrary workloads that may differ from those in t. By "workload" we mean the sequence of method invocations on the system under test and the parameters passed to these methods. For the example in Figure 2, the workload hard-codes the number of children added to the `DataNode` (one) and the name of the child ("child"). Instead, we want the checker c to be able to handle arbitrary sequences of adding and removing children with arbitrary names. Addressing this challenge requires reasoning about the meaning of magic values and domain-specific constants that appear in the test.

(2) *Generalizing assertions.* The assertions in t are specific to the workload encoded in t, but the checker c should be able to validate semantic properties under arbitrary workloads. For our example, the checker should validate that the set returned by `getChildren` is always non-null and that its size matches the number of children added minus the number of children removed, regardless of the specific sequence of additions and removals. Clearly, the assertions in t cannot be used verbatim in c as they are tied to the specific workload in t.

(3) *Reasoning about internal state.* The semantic properties encoded in t often involve the internal state of the system, which may change over time as methods are invoked. To keep track of such state, the checker c needs to maintain its own representation of the relevant internal state and update it as methods are invoked. For our example, the checker needs to track the number of children added and removed to correctly validate the size of the set returned by `getChildren`.

Prior work on deriving runtime checking mechanisms does not sufficiently address these three challenges. One line of prior work, such as Daikon [24] and Dinv [28], focuses on mining invariants from program executions. While these approaches provide a way to derive generalized assertions, they rely on a rich set of execution traces to identify patterns and invariants. Another line of work infers semantics rules, such as that every invocation of a method x implies a subsequence of invocation of another methods y [42]. However that approach requires test cases that reproduce known failures as its input, which limits its applicability. Approaches like DLint [27], DynaPyt [22], and Dylin [21] offer another perspective, by checking for common programming errors at runtime. However, they focus on generic error patterns and do not check any project-specific properties.

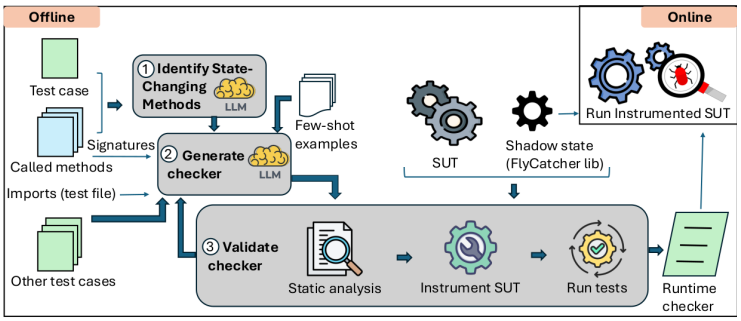


Fig. 3. Overview of FlyCatcher.

The recent T2C approach [45] is most closely related to our work, as it also focuses on generalizing test cases into runtime checkers. T2C addresses Challenges 1 and 2 via static analysis, which limits its ability to capture the intent of the tested code and the role that magic values and constants play in the test. Moreover, T2C does not address Challenge 3 because it lacks a mechanism to track and reason about the internal state of the system. Getting back to our motivating example, given the test in Figure 2, the existing approach fails to catch the bug in Figure 1. The reason is that the T2C-generated checker does not track the number of children added and removed, and instead always asserts the size to be zero, which is incorrect.

3 APPROACH

The following presents FlyCatcher, a novel approach to address the *test-to-checker* problem defined in Section 2 through a combination of LLM-based synthesis, lightweight static analysis, and dynamic validation. We start by providing an overview of the approach (Section 3.1) and then describe its key components in detail (Sections 3.2–3.6).

3.1 Overview

Figure 3 presents an overview of our approach, which consists of two phases: an *offline* phase, during which the approach infers checkers from tests, and an *online* phase, during which the checkers are executed alongside the system.

In the offline phase, FlyCatcher takes a *target test* t, a set $T_{context}$ of context tests, and a set T_{val} of validation tests from the system's test suite as input, infers a generalized runtime checker from t and validates it using T_{val}. This phase consists of three main steps, as shown in the gray boxes in Figure 3. First, FlyCatcher uses an LLM to identify those methods among all methods called in t that affect the state of the system, as well as additional methods that are called in $T_{context}$ and can affect the same system state. We call these methods the *state-changing methods*. Second, the approach uses a combination of static analysis and LLM prompting to generate a runtime checker that monitors the state-changing methods and generalizes the properties tested by t. Finally, FlyCatcher validates each runtime checker using a multi-step process, and if necessary, refines the checker by repeating step 2. The validation involves lightweight static analysis (e.g., ensuring the checker compiles and contains at least one assertion) and dynamic validation (e.g., running the checker alongside the system and ensuring that it passes all tests in T_{val}). If any validation step reports a problem, an error message is fed back to the checker generation step, and the process repeats until a successful checker is produced or a maximum of k attempts is reached.

In the online phase, the validated runtime checker is deployed alongside the system, which FlyCatcher instruments to invoke the checker whenever a state-changing method is called. When invoked, the checker observes the operation being executed and the current state of the system,

updates its own representation of the system state (the *shadow state*, see below), and asserts that the properties encoded in the checker hold. Runtime checking can be performed on any execution of the system, including executions of additional, held-out tests (as done in our evaluation) or in production.

3.2 Shadow State

A key feature of the runtime checkers generated by FlyCatcher is their ability to reason about the internal state of the system under analysis. To reason about internal system state, the approach maintains a *shadow state*, i.e., an abstraction of the system's relevant state that evolves in parallel with the real system but is isolated from it. This shadow state enables the checker to reason about and validate expected behaviors without introducing side effects or altering the actual runtime environment.

The shadow state is represented as a hierarchical mapping $S : O \rightarrow (P \rightarrow V)$, where:

- O is the set of objects in the target system,
- P is the set of property names, and
- V is the set of property values.

This representation of the shadow state allows the checker to maintain arbitrary information about the objects in the system, such as the values of their fields or other derived properties. For example, a checker derived from the test in Figure 2 may maintain, for each `DataNode` object, a property "children" that tracks the set of children added and removed via the `addChild` and `removeChild` methods. Another checker may store only the number of children, rather than the children themselves. Leaving the properties to track open-ended, instead of enforcing a specific structure, allows the inferred checkers to capture a wide range of semantic properties.

To create, maintain, and use the shadow state, checkers perform three main operations: initialization, updates, and reads. When the checker observes an object $o \in O$ for the first time, it initializes its shadow state by creating a new mapping $S(o)$ that captures the initial values $f_o : P \rightarrow V$ of the relevant properties of o. To maintain the shadow state as the system executes, the checker updates the shadow state whenever it observes a state-changing operation that affects an object o. Finally, to validate that the properties encoded in the checker hold, the checker reads values from the shadow state and compares them against the actual state of the system. For example, a checker may assert that the size of the set of children in the shadow state matches the size of the set returned by the `getChildren` method in the real system.

3.3 Identifying State-Changing Methods

The idea of the runtime checkers generated by FlyCatcher is to monitor and reason about the state of the system as it evolves over time. A crucial step toward this end is to identify the methods that can change the state of the system, as these methods are the ones that the checker needs to monitor and handle to keep its shadow state up to date. Given the target test t, FlyCatcher first analyzes the test code to identify the methods that are invoked and may have side effects. These state-changing methods are then used in the subsequent checker generation (Section 3.4), validation (Section 3.5), and instrumentation steps (Section 3.6).

Identifying state-changing methods is a non-trivial task that can be addressed in various ways. One possible approach is to statically analyze the code of the methods invoked in the test and to reason about their side effects. Prior work has proposed analyses to identify pure methods, i.e., methods guaranteed to not have any side effects [55, 57]. Such analyses typically depend on call graph analysis and points-to analysis, which are non-trivial to scale to large code bases. Another possible approach is to use heuristics based on method names, such as considering methods

Many methods called in a test may modify the state of the target system, e.g., write and delete. Given a test case and the implementation of the methods called in the test, identify all method calls that can cause side effects in the target system and produce a new version of the test containing the `// state-changing` comment in each line with method calls that can cause side effects. Constructors are state-changing methods by default.
Method implementations: [implementations]
Test: [test]

Fig. 4. Prompt for identifying state-changing methods.

with names containing words like "add", "remove", or "set" as state-changing [45]. However, this approach may miss methods with less obvious names because any hard-coded set of method names is unlikely to be comprehensive.

Motivated by the limitations of prior approaches, FlyCatcher leverages the capabilities of LLMs to identify state-changing methods based on their names and the context in which they are used. Specifically, the approach prompts an LLM with a description of the task, the source code of the methods invoked in the test, and the test code itself. To accurately identify the methods invoked by a test case, we build on the static analysis framework CodeQL [1]. Figure 4 shows the prompt template, which asks the LLM to annotate state-changing method calls with a specific comment. For our example test case in Figure 2, the approach properly identifies `DataNode`, `addChild`, and `removeChild` as state-changing operations and annotates the test case accordingly. Given the response by the LLM, we parse the test code to identify the methods annotated as state-changing.

3.4 Checker Generation

Based on the set of state-changing methods, the next step of FlyCatcher is to generate a runtime checker that monitors invocations of these methods and validates the semantic properties encoded in the target test t. Prior approaches to this problem often rely on specific templates or patterns, such as invariants expressed via binary operations between two variables [24, 28] or checks that require a call of method x to be followed by a call of method y [42]. To enable FlyCatcher to capture a wider range of semantic properties, we allow checker code to perform arbitrary computations and to include arbitrary assertions. To synthesize such checkers, FlyCatcher leverages the capabilities of LLMs to "understand" the intent of the given target test and generalize it into a runtime checker. The approach prompts the LLM with a detailed set of guidelines, few-shot examples, the target test t, and relevant information about t (e.g., import statements, context tests), as follows.

3.4.1 Guidelines and Few-Shot Examples. Because the task of generalizing test cases into runtime checkers is non-standard, it is essential to provide the LLM with detailed guidelines on the expected structure and behavior of the generated checkers. Our guidelines consist of ten requirements, which we developed and refined after observing runs of FlyCatcher on a small test benchmark:

(1) The checker is a static and parameterized method.
(2) The checker receives (i) an `Operation` object, which contains the following attributes: `signature`, `baseObject`, `arguments`, and `returnValue`; and (ii) a shadow state mapping objects to their properties and respective values.
(3) The checker handles methods that modify the state of object instances. These methods are marked with a `// state-changing` comment in the target test. For these methods, the checker updates the provided shadow state.
(4) When reading a property from the shadow state, the checker falls back to a default value, as the property may not be in the shadow state yet.

(5) The checker updates the values in the shadow state based on the semantics of the received `Operation` object.
(6) Toward the end of the checker code, the checker asserts that properties have their expected values. Similar to the assertions in the test case, the assertions may use methods that do not modify the system state. Assert statements should be outside of if-statements, as in the test. Do not insert any return statements.
(7) To obtain the expected value in the assertion, the checker retrieves it from the shadow state.
(8) The checker does not modify the state of the `baseObject` from the received `Operation`.
(9) The checker only contains necessary variables and operations, and properly accesses methods and attributes according to their visibility.
(10) The checker code is explained with comments.

To illustrate the requirements, the prompt contains five few-shot examples. We manually develop these examples, each containing a test case and its corresponding runtime checker. The few-shot examples are for simple toy classes, such as a `BankAccount`, a `ListManager` and a `Rectangle`, designed to help the LLM understand the task and the expected output format.

3.4.2 Target Test and Related Information. The most important input to the LLM is the target test t for which we want to generate a runtime checker. However, providing the target test alone may be insufficient for the LLM to understand the context and semantics of the code tested in t. For example, this happens when the class under test offers multiple methods but t exercises only a subset of them. To provide more contextual information on how the system is used beyond the target test, FlyCatcher incorporates existing context tests $T_{context}$ that also test the classes exercised by the target test t. To obtain such context tests, the approach relies on static analysis to identify all tests that invoke methods of the same classes as those invoked in t. As the number of such tests can be large, FlyCatcher randomly samples from them until exceeding a configurable number of tokens. Another important piece of information for generating a compilable checker is the import statements required by the target test t. The approach hence extracts the import statements from the test file containing t and includes them in the prompt.

Notice that FlyCatcher does not provide the implementation of the tested methods as input to the checker generation. This is to avoid that the LLM simply copies the existing implementation into the checker method. Instead, FlyCatcher relies on the semantics information conveyed in the tests only, enabling the runtime checkers to be independent of the specific implementation of the tested methods.

3.4.3 Example of Generated Checker. Figure 5 shows a runtime checker generated by FlyCatcher for the target test in Figure 2. The checker will eventually be invoked whenever one of the state-changing methods `addChild` or `removeChild` is called on a `DataNode` object. The checker receives as argument an operation, which represents the current method being invoked, and the shadow state. First, the checker gets the current state, both from the system's actual state and the shadow state, of the base object of the received operation. Then, the checker initializes the `children` value in the shadow state in case it does not yet exist. After that, the checker modifies the shadow state depending on the operation that was received as the argument. If the received operation is `addChild`, the checker adds the child to the shadow state. Similarly, if the received operation is `removeChild`, the checker removes the child from the shadow state. In the end, regardless of the received operation, the checker asserts that the values in the real execution are as expected by comparing them to the values in the shadow state.

```
public static void getChildrenChecker(Operation op, Map<Object, Map<String, Object>> shadowState)
{   DataNode baseObject = (DataNode) op.baseObject;
    Map<String, Object> objectState = shadowState.getOrDefault(baseObject, new HashMap<>());

    // Initialize children set if not present
    if (!objectState.containsKey("children")) {
        // Update the objectState from the shadow shadowState
        objectState.put("children", new HashSet<String>());
    }
    // Handle "addChild" operation, which modifies the state of baseObject
    if (op.signature.equals("org.apache.zookeeper.server.DataNode.addChild(java.lang.String)")) {
        // Update the objectState from the shadow shadowState
        Set<String> children = (Set<String>) objectState.get("children");
        children.add((String) op.arguments[0]);
        objectState.put("children", children);
    }
    // Handle "removeChild" operation, which modifies the state of baseObject
    else if (op.signature.equals("org.apache.zookeeper.server.DataNode.removeChild(java.lang.String)")) {
        // Update the objectState from the shadow shadowState
        Set<String> children = (Set<String>) objectState.get("children");
        children.remove((String) op.arguments[0]);
        objectState.put("children", children);
    }

    // Assert that the baseObject returns the correct children set
    // getChildren does not modify the state of baseObject
    Set<String> actualChildren = baseObject.getChildren();
    Set<String> expectedChildren = (Set<String>) objectState.get("children");

    // Check that children set is not null
    assertNotNull(actualChildren);
    // Check that the size matches the expected size
    assertEquals(expectedChildren.size(), actualChildren.size());
    // Check that all expected children are present
    for (String child : expectedChildren) { assertTrue(actualChildren.contains(child)); }

    // Update the shadow state
    shadowState.put(baseObject, objectState);
}
```

Fig. 5. Checker generated by FlyCatcher for the motivating example.

> When trying to [compile|instrument|execute] the provided checker, the following error happens: [error]
> Please, provide a fixed version of the provided checker to fix the error.

Fig. 6. Checker refinement prompt.

3.5 Checker Validation

LLMs, while powerful, can produce incorrect or suboptimal outputs. To improve the quality of the generated checkers, FlyCatcher uses a multi-step process to validate checkers before returning them to users. If and only if a checker passes all the validation steps, FlyCatcher outputs the checker. Otherwise, the approach uses the prompt in Figure 6 with one of the feedback messages in Table 1 to produce a fixed version of the checker. This process is repeated until a successful checker is generated or a number k of attempts is reached. The refinement follows a conversation style, where the LLM has access to the previous versions of the checker and the feedback messages, enabling the model to learn from its own mistakes. The following explains the validation steps, grouped into static and dynamic validation.

3.5.1 Static Validation of Checkers. The first group of validation steps uses lightweight static analysis to check the syntactic correctness and basic properties of the generated checker (see first part of Table 1). First, the approach ensures that the generated checker is syntactically correct and

Table 1. Validation and feedback for checker refinement.

Condition	Feedback
Static validation:	
Syntax error	Syntax error in Java code. Make sure that the checker method is indeed a single method, i.e. do not output helper methods or classes.
No assertion	The checker does not contain a call to an assertion method. Make sure to include assertions outside comments.
Non-SUT method	The system under test (SUT) does not contain the following methods: [methods signatures]. Make sure that the checker handles methods from the system under analysis rather than built-in functions or methods from the test suite.
Non-fully qualified signature	The checker handles methods without fully qualified signature: [unqualified signature]. Use fully qualified names for the method and all argument types.
Dynamic validation:	
Test failure	The following tests fail: [tests logs]. The checker should be generic and robust enough to meaningfully satisfy all test cases.
Calls state-changing method	This checker is calling a state-changing method. This is not allowed.
>30m execution	The checker is making the tests run for more than 30min.

```java
package flycatcher.checkers;

import flycatcher.util.Operation;
import flycatcher.util.ShadowState;
import org.apache.zookeeper.server.DataNode;
import java.util.*;
import static org.junit.Assert.*;

public class Checker326 {
    public static void getChildrenChecker(Operation op, Map<
           Object, Map<String, Object>> shadowState) {
        if (flycatcher.util.ShadowState.inChecker)
            throw new java.lang.RuntimeException("Checker is
                  calling a state-changing method.");
        flycatcher.util.ShadowState.inChecker = true;
        try {
            // <body of checker method generated by LLM>
        } finally {
            flycatcher.util.ShadowState.inChecker = false;
        }
    }
}
```

Fig. 7. Scaffolding (highlighted in green) added around the LLM-generated checker method.

```java
public synchronized boolean addChild(String child) {
    Boolean returnValue = null;
    try {
        // <body of the original addChild method>
        // returnValue receives the value that would be
              returned by addChild
    } finally {
         Checker326.getChildrenChecker(new Operation("org.
               apache.zookeeper.server.DataNode.addChild(java.
               lang.String)",
              this, new Object[]{child}, returnValue),
                    ShadowState.state);
    }
    return returnValue;
}
```

Fig. 8. Example of instrumentation performed by FlyCatcher.

can be parsed. If the LLM outputs a checker method that contains syntax errors or that contains multiple methods instead of a single method, the approach rejects the checker and feeds the error message back to the LLM for refinement. Second, FlyCatcher validates that the checker contains at least one call to an assertion method. The rationale is that a checker without assertions would not be useful, as it does not check any property. Third, the approach ensures that the checker handles only methods that belong to the system under analysis. This step is motivated by the observation that LLMs sometimes hallucinate methods that do not exist in the system or try to monitor built-in methods from the Java standard library. Finally, FlyCatcher checks that the checker code identifies methods using their fully qualified signature.

To turn the checker method generated by the LLM into a compilable and executable checker, FlyCatcher post-processes the generated code by integrating it into a scaffolding file. Figure 7 illustrates the scaffolding file for the checker in Figure 5. The code highlighted in green is part of the scaffolding file, while the rest is the checker code generated by the LLM. The scaffolding code maintains a flag to ensure that the checker does not call state-changing methods, which would lead to infinite recursion when the instrumented state-changing method calls the checker and the checker calls that method again. To identify and avoid this problem, FlyCatcher uses a flag called `inChecker`, which is set to `true` while the checker code is executing and set to `false` otherwise.

3.5.2 Dynamic Validation of Checkers. Once a checker passes the static validation steps, FlyCatcher performs dynamic validation to ensure that the checker behaves correctly when executed alongside the system (see second part of Table 1). The dynamic validation consists of two steps: (i) instrumenting the system under analysis to invoke the checker when one of the state-changing methods is called (explained below in Section 3.6) and (ii) running the provided validation tests T_{val} on the instrumented system.

If the checker is correct and general enough, it should not cause any of the validation tests to fail. If any of the validation tests fails, FlyCatcher rejects the checker and feeds the error message back to the LLM for further refining the checker. Beyond test failures, FlyCatcher also checks two additional conditions during the execution of the validation tests. The first condition is to ensure that the checker does not call any state-changing methods, which would lead to infinite recursion. By using the scaffolding structure illustrated in Figure 7, FlyCatcher can detect when a checker calls a state-changing method, and hence, would recurses into itself. The second condition is to ensure that the execution of the validation tests with the checker activated does not take an excessive amount of time. Specifically, FlyCatcher considers a checker invalid if the execution of the validation tests with the checker activated takes more than 30 minutes to run.

If the checker passes all dynamic validation steps, FlyCatcher outputs the checker. For our running example, the checker in Figure 5 passes all static and dynamic validation steps.

3.6 Instrumentation and Online Checking

To be able to monitor the system, FlyCatcher executes the inferred runtime checkers while the system under analysis is running. The key component to enable this online checking is to instrument the system under analysis to invoke the checker whenever one of the state-changing methods is called. The instrumentation is performed as a source-to-source transformation that wraps the body of each state-changing method into code that invokes the corresponding checker. In addition, FlyCatcher also adds to the instrumented source files the necessary import statements to use the checker and the `Operation` class, which encapsulates information about the current operation being executed, and the `ShadowState` class. The instrumentation preserves method signatures, allowing drop-in replacement of the original code.

Figure 8 shows the instrumented version of the `addChild` method called in the test in Figure 2 and handled by the checker in Figure 5. The code highlighted in green is the instrumentation added by FlyCatcher. To ensure that the checker is called even if the instrumented method raises an exception, FlyCatcher wraps the method body in a `try-finally` block. After the state-changing method returns, the instrumentation calls the checker, passing it an `Operation` object that encapsulates information about the current operation, as well as the shadow state. The Operation object contains the fully qualified signature of the state-changing method, the base object of the call, i.e `this`, and the arguments and return value of the `addChild` method.

Given the instrumented system, FlyCatcher can perform online checking by executing the system as usual. Executions can be either driven by additional tests or by real users in production. Whenever

Table 2. Target systems and tests.

System	Version	Tests				
		All	W/ SUT calls	W/ assert	Passing	Targeted
Zookeeper	3.4.11	439	406	295	249	100
Cassandra	3.11.5	3,294	2,293	1,162	255	100
HDFS	3.2.2	4,320	2,916	2,162	1,325	100
HBase	2.4.0	4,287	2,869	2,257	1,812	100
Sum		12,340	8,484	5,876	3,641	400

a checker observes a violation of the properties encoded in it, it raises an assertion error, which can cause the system to crash, be logged for later analysis or handled in real-time, depending on the needs of the user.

Getting back to our running example, recall the bug in Figure 1, which causes the `getChildren` method to return an empty set even after a child has been added. When executing the buggy version of Zookeeper along with the inferred checker in Figure 5, the checker identifies the bug after the first call to `addChild`. In this case, the `expectedChildren.size()` in the shadow state is one, but the `actualChildren.size()` in the original system is zero.

4 EVALUATION

Our evaluation addresses the following research questions:

- RQ1: How effective is FlyCatcher at inferring runtime checkers?
- RQ2: Are the checkers generated by FlyCatcher useful to detect errors?
- RQ3: Do FlyCatcher's checkers incur false positives?
- RQ4: How does feedback refinement contribute to FlyCatcher's effectiveness?
- RQ5: What are the costs of creating and using runtime checkers?

4.1 Experimental Setup

Target Systems and Tests. We apply our approach to four complex and widely-used systems, listed in Table 2. We select these systems because they are representative of real-world software, with complex functionalities and rich APIs, because they contain extensive tests suites, and because they were used to evaluate the most closely related prior work [45]. To select target tests to infer runtime checkers from, we start from all the tests in each system and apply three filtering steps that keep tests with the following properties: (i) the test calls at least one method from the target system (system under test, SUT); (ii) the test contains at least one assertion; and (iii) the test terminates and passes within a three-minute timeout. This filtering results in 3,641 tests across the four systems, from which we randomly sample 100 from each system to obtain our benchmark of 400 target tests.

Baseline. We compare FlyCatcher with T2C [45], the state-of-the-art approach to derive runtime checkers from tests. Unlike FlyCatcher, which combines LLM-based reasoning with static and dynamic analysis, T2C is based on static and dynamic analysis only. We apply their approach to the same 400 target tests as FlyCatcher.

LLMs. As the core of FlyCatcher relies on LLMs, we evaluate our approach with two different models to assess its robustness to the choice of model: Claude Sonnet 4[2], developed by Anthropic, and GPT-4o[3], developed by OpenAI.

Parameters. We set the maximum number k of attempts to fix a checker to 125. However, when the same kind of problem, e.g., a compilation error, still persists after five fix attempts in a row, we

[2]https://docs.claude.com/en/docs/about-claude/models/overview
[3]https://platform.openai.com/docs/models/gpt-4o

Table 3. Effectiveness of FlyCatcher and baseline at generating checkers.

System	Target tests	Approach	Checkers	
			Validated	Cross-Validated
Zookeeper	100	FlyCatcher (Claude Sonnet 4)	**87**	**80**
		FlyCatcher (GPT-4o)	67	61
		T2C	-	30
Cassandra	100	FlyCatcher (Claude Sonnet 4)	**78**	**77**
		FlyCatcher (GPT-4o)	63	59
		T2C	-	13
HDFS	100	FlyCatcher (Claude Sonnet 4)	**79**	**69**
		FlyCatcher (GPT-4o)	61	53
		T2C	-	18
HBase	100	FlyCatcher (Claude Sonnet 4)	**90**	**74**
		FlyCatcher (GPT-4o)	65	58
		T2C	-	27

configure FlyCatcher to stop. For each query to an LLM, we set the number of completions to one. Moreover, we limit the set $T_{context}$ of context tests to 30k tokens.

Implementation and Hardware. We implement FlyCatcher mostly in Python. To identify the static data used as input for the two first steps of FlyCatcher, we rely on CodeQL [1]. The system instrumentation is implemented in Java using the Javassist bytecode editing library. The checker generation and validation experiments are done in a server with 4-core 2.60GHz CPUs, with 15 GB memory, running Ubuntu 24.04. The remaining experiments, e.g., execution of tests to get the 400 target tests benchmark, mutation analysis, and overhead measurements, are done in a server with 48-core 2.20GHz CPUs, with 251 GB memory, running Ubuntu 22.04.

4.2 RQ1: Effectiveness

To assess the effectiveness of FlyCatcher at generating runtime checkers, we use two metrics: (i) the number of *validated* checkers, i.e., checkers that pass both static and dynamic validation; and (ii) the number of *cross-validated* checkers, i.e., validated checkers that also pass dynamic validation when executed under workloads different from those used to generate them. For cross-validation, we run the entire test suite of the target system with all validated checkers enabled.

Table 3 presents the results for FlyCatcher, both with Claude Sonnet 4 and GPT-4o, and for the T2C baseline. FlyCatcher, when used with either model, produces a substantial number of validated checkers for the target tests. The Claude Sonnet 4 model is more effective than GPT-4o, producing a total of 334 validated checkers across all 400 target tests. We do not report the number of validated checkers produced by T2C at this stage because their approach reports checkers on a per-assertion basis, whereas FlyCatcher generates one checker per test, making the numbers incomparable. The last column of Table 3 reports the number of cross-validated checkers, which is the most important metric, as it measures to what extent the checkers correctly generalize to new workloads. We find that FlyCatcher consistently and substantially outperforms the baseline, irrespective of the model used. While T2C produces only 13–30 cross-validated checkers per system, FlyCatcher with Claude Sonnet 4 yields 69–80 such checkers. In total, FlyCatcher produces 300 cross-validated checkers, which is 2.6x more than T2C.

4.3 RQ2: Usefulness for Detecting Errors

As most techniques to find bugs can find only a small subset of all bugs [32], a meaningful evaluation of FlyCatcher's ability to find bugs requires a large set of known bugs. However, real-world bugs

Table 4. Usefulness for finding errors.

Mutants covered by target tests:	
All	3,366
Killed by target tests	1,544
Survived	1,822
Killed by checkers:	
FlyCatcher	**26**
T2C	5

Table 5. Reasons why FlyCatcher misses some bugs.

Reason	Nb. cases
Limited workload	12
Equivalent mutant	18
Total	30

```
public long getOpenFileDescriptorCount() {
  Long ofdc;
  if (!ibmvendor) {
    ofdc = getOSUnixMXBeanMethod("
           getOpenFileDescriptorCount");
    return (false ? ofdc.longValue() : -1);
  }
  // ...
}
```

Fig. 9. Bug detected by FlyCatcher: The equality check is replaced with `false`, causing the method to return `-1` when `bmvendor` is false.

are relatively rare and often hard to reproduce. We thus follow the established practice [7, 58, 60] of using mutations [19, 36] to create a diverse set of known bugs and then measure how many of them our approach detects. The idea behind mutation testing is to apply small syntactic changes to the original system to create faulty programs called mutants. A mutant is said to be killed if a test case fails when executed against it. There is a strong correlation between the ability to detect mutants and real errors [37].

The computational cost of using mutation testing is high, due to the high number of generated mutants and the high computing time to execute the test suite against each mutant. To balance the costs of mutation testing, we focus on Zookeeper in this research question, as it takes "only" 48 hours to run on a machine with 40 CPU cores, which is less time than the other systems. To create mutants, we use PITest [16], a mutation testing tool for Java. We use all the mutation operators available in PITest, and apply them to all classes containing methods called in those target tests of Zookeeper where FlyCatcher infers a cross-validated checker. For each mutant that is covered by the target tests but not killed by them, i.e., a bug that would usually remain undetected, we apply the FlyCatcher-generated checkers and measure whether the mutant gets killed when running the target tests with the checkers enabled. To compare with the baseline, we also run T2C's checkers on the same set of mutants. However, as their approach is not fully automated and requires to manually configure the classes to be instrumented for each run, we apply T2C only to those mutants that FlyCatcher is able to kill.

Table 4 presents the number of mutants covered, killed, and survived by the target tests. Out of all 3,366 mutants, 1,822 survive the target tests despite being covered by them. That is, the assertions in the target tests are not strong enough to catch these mutants. Running the same target tests, but now with the 80 FlyCatcher-generated and cross-validated checkers enabled, kills 26 of these mutants. This shows that the checkers produced by FlyCatcher are indeed useful to detect errors that would otherwise remain undetected. Applying checkers generated by the T2C baseline to these 26 mutants, we find that those checkers kill only 5 of them. That is, FlyCatcher detects 5.2x more mutants than T2C, which is a substantial improvement. To further put these numbers in perspective, we refer to a study showing that static bug detectors, i.e., a class of bug detection tools that is widely used in practice, typically find between 1%–3% of all bugs in a system [32].

Examples of detected and undetected mutants. The faulty version of the `getChildren` method in Figure 1 is one of the mutants generated by PITest with the *Empty returns* mutator, which is detected by FlyCatcher. Figure 9 shows another mutant detected by FlyCatcher, where the buggy code never returns the result of evaluating `ofdv.longValue()`, as the condition in the ternary expression is `false`. FlyCatcher produces a checker that monitors the values returned by `getOpenFileDescriptorCount()` and is able to detect this bug.

To better understand the limitations of our approach, we manually inspect 30 of the mutants that are not killed by FlyCatcher. Table 5 shows the two reasons we observe for why FlyCatcher misses

```
public boolean getUnix() {
  if (false) {
    return false;
  }
  return (ibmvendor ? linux : true);
}
```

(a) The condition `Windows` is replaced with `false`.

```
/**
* these are the number of acls that we
      have in the datatree
*/
- long aclIndex = 0;
+ long aclIndex;
```

(b) Equivalent mutant where the assignment to `aclIndex` is removed.

```
// let's be conservative on the typical
      number of children
children = new HashSet<String>(9);
```

(c) Equivalent mutant where the HashSet size argument changes from 8 to 9.

Fig. 10. Examples of mutants missed by FlyCatcher.

Table 6. Analysis of false positives caused by FlyCatcher's checkers.

System	Activated checkers	Checker calls	Checker failures	False positive
Zookeeper	44%	5,509,724	169	0.00003
Cassandra	37%	37,600	0	0

these bugs. 12 of the inspected cases are due to *Limited workload*, which means that the existing tests do not trigger the buggy path. We rely on the existing tests in Zookeeper to run our checkers on the mutants, which may not cover all possible scenarios. Additionally, the environment we use for our experiments also may not cover all possible scenarios. The mutant in Figure 10a is not detected as our experiments are executed on Linux, and the condition to check for Windows is replaced with false. The second reason we found for FlyCatcher missing some bugs is *Equivalent mutants*, which accounts for the other 18 cases. An equivalent mutant is a mutant that is syntactically different from the original program but semantically equivalent. Figure 10b shows an example of an equivalent mutant missed by FlyCatcher. In this case, the initial assignment to the long variable is removed. However, in Java, long variables receive zero as default value. Figure 10c shows another case we found in this category. The mutant modifies the argument passed when initializing a `HashSet`. However, this does not change the functional correctness of the program, as the argument determines only the initial capacity of the set, which can grow dynamically.

4.4 RQ3: False Positives

To measure the false positives of FlyCatcher's checkers in real executions, we follow the steps to calculate false positives done in T2C: run the system under analysis on five nodes using Jepsen [2], a widely used testing framework for distributed systems. Jepsen generates common workloads and automatically injects network faults every 10 seconds. As setting up this experiment requires non-trivial configuration efforts, we focus on the two systems for which FlyCatcher generates the highest number of cross-validated checkers: Zookeeper and Cassandra. For each checker, we run the system with Jepsen for five minutes. Table 6 shows the results. In Zookeeper, 44% of the 80 checkers are activated and, in total, called over five million times, from which only 169 calls failed. In Cassandra, 37% of the 77 checkers are activated and, in total, called 37,600 times, all of which succeed. We conclude that FlyCatcher produces checkers with very few false positives, making the checkers suitable for use in practice.

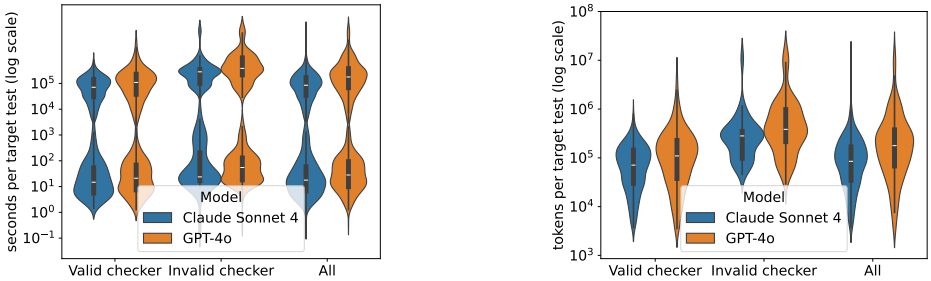
4.5 RQ4: Ablation Study on Feedback-Based Refinement

A core component of FlyCatcher is the feedback-based refinement of checkers, which leverages both static and dynamic validation to iteratively fix invalid checkers (Section 3.5). We evaluate the contribution of each validation step to FlyCatcher's overall effectiveness. To this end, we measure the number of validated checkers generated by FlyCatcher under different feedback configurations for all 400 target tests drawn from the four systems, using both models.

The results are summarized in Table 7. Without any feedback, FlyCatcher produces only 48 validated checkers with Claude Sonnet 4 and 18 with GPT-4o. Incorporating static validation feedback

Table 7. Effectiveness of FlyCatcher with different kinds of feedback provided to refine checkers.

System	Target tests	Approach	Validated checkers	
			Claude Sonnet 4	GPT-4o
Zookeeper	100	FlyCatcher w/o feedback	8	9
		FlyCatcher w/ static validation	49	37
		FlyCatcher w/ dynamic validation	**87**	**67**
Cassandra	100	FlyCatcher w/o feedback	12	4
		FlyCatcher w/ static validation	47	26
		FlyCatcher w/ dynamic validation	**78**	**63**
HDFS	100	FlyCatcher w/o feedback	12	2
		FlyCatcher w/ static validation	48	33
		FlyCatcher w/ dynamic validation	**79**	**61**
HBase	100	FlyCatcher w/o feedback	16	3
		FlyCatcher w/ static validation	46	35
		FlyCatcher w/ dynamic validation	**90**	**65**



(a) Time. (b) Tokens.

Fig. 11. Distribution of costs to generate a runtime checker with FlyCatcher.

significantly improves effectiveness, yielding 190 and 131 validated checkers with Claude Sonnet 4 and GPT-4o, respectively. Finally, dynamic validation feedback further improves the effectiveness, resulting in 334 and 256 validated checkers for Claude Sonnet 4 and GPT-4o, respectively. These findings demonstrate that the feedback mechanism in FlyCatcher is critical to its effectiveness, regardless of the underlying model.

4.6 RQ5: Costs of the Approach

We measure the following costs imposed by FlyCatcher to produce a runtime checker: (i) time to produce a runtime checker; (ii) number of tokens consumed by queries to the LLM; and (iii) monetary costs associated with the token consumption, based on OpenAI's and Anthropic's pricing as of October 2025. In addition, to assess cost of using the checkers, we also measure (iv) runtime overhead incurred when running the checkers.

Figure 11 presents the costs to generate a checker per target test case. The median time to generate a checker that is successfully validated is 15 seconds with Claude Sonnet 4 and 21 seconds with GPT-4o. However, for some outliers, the checker generation process can take several hours. The upper "belly" in the distributions in Figure 11a is from Cassandra, a particularly slow system in our evaluation. Considering token costs, FlyCatcher with Claude Sonnet 4 consumes 178k input tokens and produces 5k output tokens, on average per target test. This corresponds to a total of 183k tokens and a monetary price of USD 0.60 per target test. With GPT-4o, FlyCatcher consumes 552k

input tokens and produces 9k output tokens. This is a total of 561k tokens and a monetary price of USD 1.8 per test. These results demonstrate that a more effective model tends to produce more valid checkers in less time, whereas a less effective model keeps trying to refine invalid checkers, consuming more resources.

To measure the runtime overhead imposed by using checkers produced by FlyCatcher, we run the test files containing the target tests that FlyCatcher was able to produce validated checkers for. We run each test file five times with and without the checkers active in the system to account for variability in execution time and report average times. We apply this process to three systems: Zookeeper, Cassandra and HBase. For Zookeeper, the average execution time without and with the checkers is 1,139 and 1,170 seconds, respectively. That is, the relative overhead is only 2.7%. For Cassandra, the average execution time without and with the checkers is 104 and 122 seconds, respectively. That is a relative overhead of 17.3%. For HBase, the average execution time without and with the checkers is 4,308 and 6,041 seconds, respectively. That is, we observe a relative overhead of 40.3%. It is important to note that these overhead results can be seen as an upper bound, as we run only the test files containing the target tests, which are a small subset of all tests in the system. When running the entire test suite, or a production run of the target systems, the overhead is expected to be lower, as most tests do not have checkers enabled.

5 THREATS TO VALIDITY

A first limitation concerns the generalizability of our evaluation. FlyCatcher is assessed on four large, well-tested Java systems, which provides a diverse yet ultimately narrow sample of software. The approach may behave differently on other programming languages, smaller projects, or systems with fewer or qualitatively different tests. Moreover, the random sampling of tests and context tests introduces potential bias, as some types of assertions or code paths may be over- or under-represented. To mitigate such bias, we randomly sample a relatively large number of tests (4x100).

Another threat lies in the evaluation methodology. The bug-finding experiment relies on mutation testing as a proxy for real defects and uses only a single subject system, which limits the conclusions that can be drawn about real-world effectiveness. While mutants provide a controlled setting to measure fault detection, they may not fully reflect realistic fault distributions or semantics. Furthermore, the approach depends on large language models whose outputs vary with model version, temperature, and prompt phrasing, introducing nondeterminism. We mitigate this threat by documenting model versions and prompts, and by sharing detailed logs of our experiments.

6 RELATED WORK

Silent failures. Modern software systems experience increasingly complex failure patterns [3, 5, 26, 35, 46, 51, 53, 69, 70], such as partial failures [41], fail-slow faults [31, 47, 68], metastable failures [34]. Unlike failures that raise explicit error signals, silent failures [44, 62] lack clear indicators and are therefore difficult to detect. Lou et al. conduct a study [42] on silent failures in distributed systems, confirming that silent failures occur frequently and have significant consequences in practice.

Runtime checking. To detect silent failures, researchers have proposed frameworks [6, 10, 14, 29, 30, 39, 40, 50, 54, 59] that enable developers to conveniently specify semantic properties and verify them at runtime. Early work enables developers to write parametric specifications and translates them into AspectJ-checkable formats [12] and enables to check type-state properties within a fixed overhead budget [6]. Recent work introduces a new specification languages [18] and extends runtime checking to new domains and languages [21, 22] A drawback of these approaches is that they require developers to manually write specifications, which is time-consuming and error-prone. Instead, FlyCatcher automatically generates runtime checkers from test cases, i.e., a resource available for most complex software.

Invariant mining. Manually writing specifications is time-consuming and error-prone. A substantial body of work [4, 9, 11, 15, 17, 33, 38, 48, 66] has explored automatically mining likely invariants from software execution traces. Early systems, such as Daikon [23] and Dinv [28], infer key state relationships from test executions. The more recent Oathkeeper [43] work extracts semantic relations between events by comparing test runs of buggy and patched versions. These systems often suffer from high inaccuracy due to their statistical nature, and their inferred invariants are often not expressive enough to capture the full richness of a system's semantics. Our work differs in three key aspects. First, we do not rely on execution traces, but instead leverage the source code of tests. Second, instead of extracting invariants or patterns that fit specific templates, FlyCatcher-generated checkers can contain arbitrary logic and state representations. Finally, our approach reasons via LLMs about semantic properties encoded in tests, enabling it to generalize beyond the specific workloads and assertions encoded in the tests.

Test transformation. Test cases serve as a valuable resource for discovering underlying program semantics and generating checkers. Existing works such as PCheck [65], Ctests [61] and ZebraConf [49] transform tests to executable checks to examine validity and correctness of configurations. In contrast, our work focuses on checking runtime behaviors instead of static configuration values.

LLM-based reasoning and code generation. The use of LLMs to reason about software and generate code has rapidly become a popular trend among researchers and developers [8, 13, 52, 56, 63, 64]. TiCoder [25] proposes an interactive workflow to partially formalize natural language guides and generates more accurate code. ClassEval [20] proposes a new benchmark and evaluates 11 state-of-the-art LLMs on class-level code generation. KNighter [67] automatically synthesizes static analyzers from historical bug patterns to expose new bugs. Our system targets at runtime checker generation and deals with distinct challenges such as minimizing runtime safety risks.

7 CONCLUSION

This paper introduces FlyCatcher, a novel approach that automatically infers semantic runtime checkers from existing tests by combining LLM-based synthesis, static analysis, and dynamic validation. By leveraging large language models to generalize the intent behind test assertions and maintaining a shadow state for reasoning about system behavior, FlyCatcher generates stateful and robust checkers that extend beyond the specific workloads encoded in tests. Our evaluation on four complex software systems shows that FlyCatcher produces substantially more correct and general checkers than prior work and detects significantly more otherwise missed errors. These results demonstrate that LLM-guided synthesis, coupled with iterative validation, can transform existing tests into powerful runtime checkers, thereby enabling the broader and more practical use of runtime monitoring to detect silent failures in real-world software.

8 DATA AVAILABILITY

The code and data associated with our work is available: https://github.com/biabs1/FlyCatcher.

REFERENCES

[1] 2026. CodeQL. https://codeql.github.com/ Accessed: 2026-01-29.
[2] 2026. Jepsen: Distributed Systems Safety Research. https://jepsen.io/ Accessed: 2026-01-29.
[3] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 390–408. https://www.usenix.org/conference/osdi18/presentation/alagappan
[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 4–16. https://doi.org/10.1145/503272.503275

[5] George Amvrosiadis and Medha Bhadkamkar. 2016. Getting Back Up: Understanding How Enterprise Data Backups Fail. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 479–492. https://www.usenix.org/conference/atc16/technical-sessions/presentation/amvrosiadis

[6] Matthew Arnold, Martin T. Vechev, and Eran Yahav. 2008. QVM: An efficient runtime for detecting defects in deployed systems. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 143–162.

[7] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. 2008. On the Effectiveness of Manual and Automatic Unit Test Generation. In *Proceedings of the Third International Conference on Software Engineering Advances, ICSEA 2008, October 26-31, 2008, Sliema, Malta*. IEEE Computer Society, 252–257. https://doi.org/10.1109/ICSEA.2008.66

[8] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 85–111. https://doi.org/10.1145/3586030

[9] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 468–479. https://doi.org/10.1145/2568225.2568246

[10] Jacob Burnim, Tayfun Elmas, George C. Necula, and Koushik Sen. 2011. NDSeq: runtime checking for nondeterministic sequential specifications of parallel correctness. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 401–414.

[11] Jacob Burnim and Koushik Sen. 2010. DETERMIN: inferring likely deterministic specifications of multithreaded programs. In *International Conference on Software Engineering (ICSE)*. ACM, 415–424.

[12] Feng Chen and Grigore Rosu. 2007. MOP: An efficient and generic runtime verification framework. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 569–588.

[13] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 674–688. https://doi.org/10.1145/3627703.3629553

[14] Alvin Cheung and Samuel Madden. 2008. Performance profiling with EndoScope, an acquisitional software monitoring framework. *Proc. VLDB Endow.* 1, 1 (aug 2008), 42–53. https://doi.org/10.14778/1453856.1453866

[15] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining specifications of malicious behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 5–14. https://doi.org/10.1145/1287624.1287628

[16] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 449–452. https://doi.org/10.1145/2931037.2948707

[17] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 281–290. https://doi.org/10.1145/1368088.1368127

[18] Joshua Heneage Dawes and Domenico Bianculli. 2024. Checking complex source code-level constraints using runtime verification. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 255–265.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection help for the practicing programmer. *IEEE Computer* 11, 4 (April 1978), 34–41.

[20] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating Large Language Models in Class-Level Code Generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. https://doi.org/10.1145/3597503.3639219

[21] Aryaz Eghbali, Felix Burk, and Michael Pradel. 2025. DyLin: A Dynamic Linter for Python. *Proc. ACM Softw. Eng.* 2, FSE (2025), 2828–2849. https://doi.org/10.1145/3729395

[22] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[23] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/302405.302467

[24] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 213–224.

[25] 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 Trans. Softw. Eng.* 50, 9 (Sept. 2024), 2254–2268. https://doi.org/10.1109/TSE.2024.3428972

[26] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 149–166. https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

[27] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

[28] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*. 1149–1159.

[29] Kevin Guan, Marcelo d'Amorim, and Owolabi Legunsen. 2025. Faster Explicit-Trace Monitoring-Oriented Programming for Runtime Verification of Software Tests. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 405 (Oct. 2025), 30 pages. https://doi.org/10.1145/3763183

[30] Kevin Guan and Owolabi Legunsen. 2025. TraceMOP: An Explicit-Trace Runtime Verification Tool for Java. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 1218–1222.

[31] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=3189759.3189761

[32] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *ASE*. ACM, 317–328.

[33] Sudheendra Hangal and Monica S. Lam. 2002. Tracking down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida) *(ICSE '02)*. Association for Computing Machinery, New York, NY, USA, 291–301. https://doi.org/10.1145/581339.581377

[34] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[35] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*. ACM, British Columbia, Canada, 7 pages.

[36] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678.

[37] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 654–665. https://doi.org/10.1145/2635868.2635929

[38] Choonghwan Lee, Feng Chen, and Grigore Roşu. 2011. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 591–600. https://doi.org/10.1145/1985793.1985874

[39] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. Projection-based runtime assertions for testing and debugging Quantum programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 150:1–150:29. https://doi.org/10.1145/3428218

[40] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*. USENIX Association, Cambridge, MA.

[41] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 559–574. https://www.usenix.org/conference/nsdi20/presentation/lou

[42] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 91–107.

[43] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. Carlsbad, CA, 91–107.

[44] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. 2025. Deriving semantic checkers from tests to detect silent failures in production distributed systems. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 19–38.

[45] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. 2025. Deriving Semantic Checkers from Tests to Detect Silent Failures in Production Distributed Systems. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, Lidong Zhou and Yuanyuan Zhou (Eds.). USENIX Association, 19–38. https://www.usenix.org/conference/osdi25/presentation/lou

[46] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 114–130. https://doi.org/10.1145/3341301.3359645

[47] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. 2025. One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation* (Philadelphia, PA, USA) *(NSDI '25)*. USENIX Association, 359–378. https://www.usenix.org/conference/nsdi25/presentation/lu

[48] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 103–116.

[49] Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. 2021. Finding heterogeneous-unsafe configuration parameters in cloud systems. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 410–425. https://doi.org/10.1145/3447786.3456250

[50] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

[51] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. 2006. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* (San Jose, CA) *(NSDI'06)*. USENIX Association, USA, 17.

[52] Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. 2025. Flow-of-Action: SOP Enhanced LLM-Based Multi-Agent System for Root Cause Analysis. In *Companion Proceedings of the ACM on Web Conference 2025* (Sydney NSW, Australia) *(WWW '25)*. Association for Computing Machinery, New York, NY, USA, 422–431. https://doi.org/10.1145/3701716.3715225

[53] Shangshu Qian, Wen Fan, Lin Tan, and Yongle Zhang. 2023. Vicious Cycles in Distributed Software Systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 91–103.

[54] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. 2022. Debugging the OmniTable Way. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 357–373. https://www.usenix.org/conference/osdi22/presentation/quinn

[55] Atanas Rountev. 2004. Precise identification of side-effect-free methods in Java. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 82–91.

[56] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring LLM-Based Agents for Root Cause Analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 208–219. https://doi.org/10.1145/3663529.3663841

[57] Alexandru Sălcianu and Martin Rinard. 2005. Purity and side effect analysis for Java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 199–215.

[58] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C. Gall, and Alberto Bacchelli. 2019. On the effectiveness of manual and automatic unit test generation: ten years later. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 121–125. https://doi.org/10.1109/MSR.2019.00028

[59] Zhuohang Shen, Mohammed Yaseen, Denini Silva, Kevin Guan, Marcelo d'Amorim Junho Lee and, and Owolabi Legunsen. 2025. A Generic and Efficient Python Runtime Verification System and its Large-scale Evaluation.

[60] Beatriz Souza and Patrícia D. L. Machado. 2020. A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation. In *34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*, Everton Cavalcante, Francisco Dantas, and Thaís Batista (Eds.). ACM, 253–262. https://doi.org/10.1145/3422392.3422407

[61] 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)*. USENIX Association, 735–751. https://www.usenix.org/conference/osdi20/presentation/sun

[62] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 2023. Understanding Silent Data Corruptions in a Large Production CPU Population. In *Proceedings of the 29th Symposium on Operating Systems Principles* (<conf-loc>, <city>Koblenz</city>, <country>Germany</country>, </conf-loc>) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 216–230. https://doi.org/10.1145/3600006.3613149

[63] Yifan Wang and Kenneth P. Birman. 2025. Diagnosing and Resolving Cloud Platform Instability with Multi-modal RAG LLMs. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (World Trade Center, Rotterdam, Netherlands) *(EuroMLSys '25)*. Association for Computing Machinery, New York, NY, USA, 139–147. https://doi.org/10.1145/3721146.3721958

[64] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) *(CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 4966–4974. https://doi.org/10.1145/3627673.3680016

[65] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 619–634. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu

[66] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 117–132. https://doi.org/10.1145/1629575.1629587

[67] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3731569.3764827

[68] Andrew Yoo, Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. 2021. Fail-slow fault tolerance needs programming support. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 228–235. https://doi.org/10.1145/3458336.3465299

[69] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 575–589. https://www.usenix.org/conference/nsdi20/presentation/zhai

[70] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 116–131. https://doi.org/10.1145/3477132.3483577