REDII: Test Infrastructure to Enable Deterministic Reproduction of Failures for Distributed Systems

Yang Feng*, Zheyuan Lin*, Dongchen Zhao*, Mengbo Zhou*, Jia Liu*‡, James A. Jones†‡
*State Key Laboratory for Novel Software Technology, Nanjing University
fengyang@nju.edu.cn, {zheyuanlin, dongchenzhao, mengbozhou}@smail.nju.edu.cn, liujia@nju.edu.cn
†University of California, Irvine, California, USA, jajones@uci.edu

***Abstract*—Despite the fact that distributed systems have become a crucial aspect of modern technology and support many of the software systems that enable modern life, developers experience challenges in performing regression testing of these systems. Existing solutions for testing distributed systems are often either: (1) specialized testing environments that are created specifically for each system by its development team, which requires substantial effort for each team, with little-to-no sharing of this effort across teams; or (2) randomized injection tools that are often computationally expensive and offer no guarantees of preventing regressions, due to their randomness. The challenge of providing a generalized and practical solution to trigger bugs for reproducing and demonstrating failures, as well as to guard against regressions, is largely unaddressed.**

In this work, we present REDII, an infrastructure for supporting regression testing of distributed systems. REDII contains a dataset of real bugs on common distributed systems, along with a generalizable testing framework REDIT that enables developers to write tests that can reproduce failures by providing ways to deterministically control distributed execution. In addition to the real failures in REDII from multiple distributed systems, REDIT provides a reusable, programmable, platform-agnostic, deterministic testing framework for developers of distributed systems. It can help automate the running of such tests, for both practitioners and researchers. We demonstrate REDIT with 63 bugs that we selected in JIRA on 7 large and widely used distributed systems. Our case studies show that REDII can be used to allow developers to write tests that effectively reproduce failures on distributed systems and generate specific scenarios for regression testing, as well as providing deterministic failure injection that can help developers and researchers to better understand deterministic failures that may occur in distributed systems in the future. Additionally, our studies show that REDII is efficient for real-world system regression testing, providing a powerful tool for developers and researchers in the field of distributed-system testing.

***Index Terms*—Distributed Systems, Infrastructure, Regression Testing**

I. INTRODUCTION

In the big-data and cloud-computing era, large software companies increasingly use distributed clusters to host, process, and analyze tremendous amounts of data. These distributed systems in the cloud comprise thousands of commodity machines and use complex algorithms to maintain consistency. They are complicated with vast amounts of non-determinism (*e.g.,* parallelism from thousands of processes and threads, unforeseen network and node failures), making it difficult to understand and reproduce failures.

‡ Jia Liu and James A. Jones are the corresponding authors.

For traditional software, developers who fix bugs often write new tests and run the new and existing test cases to ensure that the bug was fixed, as well as the changes did not reintroduce known prior issues [1]. These new test cases can be used in two key ways: (1) they can be run on the *unfixed* version to reproduce the failure caused by the bug, and thus provide documentation of the runtime impact of the bug; and (2) they can be run on the *fixed* version to demonstrate that the bug fix is effective. However, for distributed systems, just writing test cases that reproduce failures is often a much more difficult task than for monolithic systems [2]. When our regression testing starts, we need a solution that can launch multiple nodes or services that compose the distributed system, as part of the test framework, so tests can be independent and controllable, which is difficult in the development environment [3], [4]. We also need a way to reproduce the environmental conditions that are often the source of many distributed-system failures, such as timings of events caused by different, parallel tasks running on different nodes in the system. Environmental failures (*e.g.,* network failures) also often cause disruptions that reveal a lack of robustness in the software, and such a test framework needs to have the ability to deterministically introduce them precisely according to the developer's will. A test framework that can allow developers to deterministically reproduce failures for distributed systems, where there is a dependence on fallible and changing environmental conditions and non-deterministic computational timings are inherent to the distributed nature of these systems, needs to be able to control for these factors.

Further, there is a relative paucity of research on regression testing for distributed systems, which may largely stem from the absence of experimental controls, subject programs, and bugs, as well as the ability to reliably and deterministically reproduce non-deterministic failures. Thus, to enable future software-engineering research on the important and pervasive domain of testing distributed systems, it is essential to have an infrastructure of well-known distributed-systems software with existing bugs. Additionally, such an infrastructure needs to ensure the deterministic reproduction of failures in a way that can provide control over timing, sequencing, and event injection. A framework that can strongly support deterministic bug regression testing of distributed systems becomes a necessity for both research and practice of software engineering.

The research approaches that exist for testing distributed systems can largely be categorized as: (1) focusing only on

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00244

the deployment of the system in the test environment (*e.g.,* [5], [6]), (2) focusing on a specific type of failure (*e.g.,* network partitioning) (*e.g.,* [4], [7]), (3) using randomness to inject failures (*e.g.,* [8]), (4) automatically injecting defects (*e.g.,* [9]–[12]), or (5) using model-based approaches to explore the state-space of failure injection as much as possible (*e.g.,* [13]–[15]). While research that focus only on a single stage or on a single bug-injection type are helpful for solving some specific issues, they unfortunately fail to offer a general and comprehensive solution for regression testing. Though it is good for bug identification, random failure injection could lead to flaky test cases. Automated failure injection and model-based approaches can also help bug identification and may be well suited for before-release comprehensive testing, but they are time-consuming and not ideal for regression testing scenarios. We cannot and should not simply migrate tools dedicated to finding bugs to a distributed-system regression testing scenario.

Moreover, the current practice for testing different distributed systems is that each development organization creates its own custom-made distributed-testing framework and testing tool libraries. Large-scale distributed systems, such as Cassandra [16], Hadoop [17], and HBase [18] each has its own regression-testing framework. Some small distributed systems only test code at the method level or connect to physical clusters for testing. They simulate real-world environments with *pseudo*-distributed designs. Even though all the mentioned frameworks have common portions, each is tightly coupled to the system for which they are designed. However, these works fall short of offering the requisite infrastructure due to their inherent lack of deterministic failure reproduction and regression testing. Furthermore, they rarely support the deterministic reproduction of environmental bugs, which is a critical aspect in understanding and mitigating such issues. Some frameworks to support regression testing [19]–[21] of distributed systems have gradually emerged, but they only provide frameworks, and have poor support for determinism in regression testing, which affects the effectiveness and efficiency of testing.

In this work, we propose the first infrastructure for the deterministic reproduction and regression testing of distributed-systems failures, namely REDII (short for **Re**gression testing for **Di**stributed systems **I**nfrastructure). REDII provides both a *dataset* REDID (short for . . . **D**ataset) of well-known distributed-systems bugs and a *toolset* for deterministically reproducing the failures caused by those bugs, called REDIT (short for . . . **T**oolset). To clarify, REDIT is not an automated tool to hunt for unknown bugs, which would be an inefficient and time-consuming process for enabling the creation and execution of regression tests. Instead, it aims to offer a reusable testing framework suitable for various systems, enabling developers to write efficient and deterministic tests. REDIT can serve as "JUnit for distributed systems" which allows for true distributed-systems testing, with facilities to deterministically control environmental conditions, such as timings, network failures. REDIT makes it easier for developers to design test cases and reproduce detected bugs given specific event orders in the regression scenarios. This distinguishes itself from the current methods that rely on custom-made distributed-testing frameworks, bug-hunting tools and random failure-injection tools. This infrastructure can be used to assist the testing of distributed systems, and could also benefit the research of distributed system testing techniques.

The contributions can be summarized as follows:

- An infrastructure, namely REDII, containing a rich bug dataset and testing framework to support the research and controlled experimentation with testing techniques of distributed systems. To the best of our knowledge, this is the first infrastructure to support deterministic reproduction of distributed-systems failures.
- A dataset, called REDID, contains 63 real bugs found across 7 distributed systems. For each bug, we prepare corresponding switches as well as tests, and thus make them controllable in the experiment of future research on tests of distributed systems, facilitating a more systematic approach to study and resolve bugs in distributed environments.
- A lightweight and cross-platform regression-testing framework for distributed systems, namely REDIT, to assist software-engineering practice and research. REDIT encapsulates typical events, which often happen in the running of distributed systems. Additionally, we have integrated three essential components, *i.e.,* an *environment engine*, an *event instrumenter*, and a *runtime system*, to ease the users to write tests for distributed systems with deterministic event orders.

Data Availability. All information on the dataset, framework, as well as documents, are open-sourced and released at [22].

II. Background and Motivation

A. Deterministic Failure-Reproduction Regression Testing

One common practice for regression testing is that a developer creates a test case at the same time as a bug fix and simultaneously commits the fix and its test case to the version-control system. This new test case is intended to provide two main characteristics. In the *pre-fixed version*, this new test case deterministically fails, which provides documentation and a demonstration of how the bug in the code leads to a failure at runtime. In the *post-fix version*, this test case demonstrates that the bug was indeed fixed. Moreover, and perhaps more importantly, it provides a safeguard against future regressions for that bug or perhaps similar bugs.

While this practice of creating a test case that deterministically reproduces the failure in the pre-fix version is common for traditional software development, it is actually quite challenging for many bugs in distributed-systems software. This challenge results from the nature of many of the most pernicious bugs in distributed software. For distributed systems, many of the most challenging bugs arise from non-deterministic environmental conditions that the software should be robust enough to handle. For example, because distributed systems are composed of multiple processes running on multiple computers over a network, problems often arise from the issues involved in parallel execution (*e.g.,* event

timings and runtime dependencies, clock drift) and issues regarding the unreliable nature of the network (*e.g.,* network partitionings or failures). Developers of distributed systems know that they need to design their software to properly handle such conditions, but it is often difficult to foresee all such possible problems in implementing all safeguards, defensive programming, timing locks, redundant data stores, and so on.

Bugs that are the result of missing all such safeguards and proper handling of these conditions are often difficult to reliably reproduce as runtime failures, because they depend upon those non-deterministic conditions replaying in the same way that the original bug reporter describes. As such, once a developer is able to discover the problem, they may want to have the ability to artificially recreate those conditions to test whether their understanding of the bug's behavior is correct. Moreover, once they are able to artificially recreate these conditions, they can write a bug fix and test that that bug fix indeed fixes the runtime failures. However, the solutions toward these goals that developers have available today are largely inadequate.

B. Practices of Distributed-System Testing

Today, a number of techniques are employed by software developers for testing distributed systems. Unfortunately, many of them cannot reliably reproduce specific environmental conditions that cause bugs to manifest as runtime failures.

Test Beds. A common practice in industry is to create a set of test servers on which to first deploy and run tests. This testing solution offers isolation from actual users to prevent side effects and allows for real, parallel distribution and testing of the most common runtime conditions. It does not, however, provide a general way to recreate environmental conditions or failures. Although such features could be built into such a test environment, each such solution would be bespoke and require each team to create their own solution. Moreover, the test cases, themselves, do not have the ability to specify their own required environmental conditions. Their environments are often shared across tests, making it challenging to perform disruptive tasks like node crashes without impacting others.

Thread-based Pseudo Distribution. Another common practice for testing distributed systems is using threading to simulate distributed, parallel execution [23]. For example, Hadoop's regression tests and Cassandra's testing tool CCM [24] both build pseudo-distributed clusters locally for their testing. Elasticsearch's testing framework even runs multiple nodes instances in a single JVM [25]. These techniques are limited by thread concurrency and are essentially *pseudo-distributed testing*. The advantage of this pseudo-distributed testing is that it has high efficiency, uses fewer resources, and does not require a solution to deploy multiple nodes on different computers. But its shortcomings are also obvious: each project requires a custom-built, per-project testing framework, leading to programming challenges and weaker testing outcomes. Most importantly, tests run in the same process and thus cannot accurately reproduce situations that exist in realistic environments, particularly with network communication (*e.g.,* enforcing specific sequences of events or network failures).

Random Failure Injection. In research, a category of testing solutions has been created to perform random environmental failure injection. Tools like Jepsen [26] and CoFI [27] have been proposed to randomly create runtime environmental conditions in order to check if the code under test properly handles them. They achieve this effect by creating a simulated distributed environment using containerization (*e.g.,* Docker). Such solutions can help discover rare timing-related issues. However, they are inherently non-deterministic, and thus do not reproduce the same environmental conditions for each run. They work well to discover triggering scenarios, but they are not designed to reproduce a specific scenario, *e.g.,* one that has been discovered and reported in a bug report.

C. Motivation

Taking inspiration from the prior category and traditional test environments such as JUnit, we envision a testing environment that allows each test case's code to specify its deployment architecture and environmental runtime requirements, such as network partitions, clock drift, event timings, and so on. Such a solution would provide developers with more granular control over the test environment, allowing them to create more realistic, controllable and reproducible test scenarios. It also could be realized through the use of containerization so that the test code can communicate with the test environment to specify runtime environmental conditions, and as such, can be used to deterministically reproduce failures.

Further, recreating the deployment environment is a challenging task for the testing of distributed systems. Currently, the most widely used method is unit testing or thread-based simulation, and thus it cannot detect some fatal failures caused by bugs that are triggered by specific environmental failures in the real production environment. Therefore, a testing environment that closely simulates the real distributed system runtime environment can effectively overcome the shortcomings of unit tests and thread simulation tests, and significantly improve the testing efficiency of distributed systems in practice.

Moreover, existing datasets of distributed-systems bugs used in research often fail to address deterministic regression testing scenarios. While these datasets cover a broad range of general use cases, they frequently overlook specific scenarios with unique constraints. As a result, researchers and developers lack suitable datasets for evaluating the effectiveness of their own methodologies and frameworks in deterministic regression testing, making it difficult to reproduce bugs in specialized contexts. To address this gap, selecting real-world bugs from distributed systems and collecting corresponding test cases that can trigger these bugs is crucial. Such a dataset would enable researchers to reproduce and analyze real bugs, thereby advancing the experimentation, development, understanding, and evaluation of novel software engineering techniques, such as automated testing, defect localization, and automated repair.

III. INFRASTRUCTURE: REDII

To support research on distributed-systems testing, we construct a systematic infrastructure containing a set of reproducible real-world bugs and a framework aimed at addressing these challenges. All these bugs are selected from real projects and can be seeded into the source code with the bug-fix code lines and files. Further, to facilitate deterministic bug reproduction, testing, and controlled experimentation by researchers and developers, we design and implement a testing framework to support the deterministic event injection and runtime-environment simulation. Our infrastructure and its components are named as such:

- **RediI**: **Re**gression testing for **di**stributed systems **I**nfrastructure
  - **RediD**: **Re**gression testing for **di**stributed systems **D**ataset
  - **RediT**: **Re**gression testing for **di**stributed systems **T**oolset

Although this is an oversimplification, one way of conceptualizing REDID is "Defects4J [28], for distributed systems," and REDIT is "JUnit with deterministic environmental controls over a simulated network environment." In the dataset, we define macro switches to toggle between the buggy and fixed versions of the code, facilitating precise control and future experimentation. To simplify the testing process, we implement a test case for each seeded bug with REDIT. Each of these test cases utilizes the REDIT for defining key components, such as deployment parameters, events, and their sequences. Additionally, for every test case implemented in REDIT, we systematically capture and store intermediate process data, including scripts, node details, runtime logs, execution commands, and results. REDII provides a framework and replicable dataset for the research of regression testing for distributed systems, rather than developing automated bug-detection tools or theoretical models. We depict our workflow and architecture of REDII in Figure 1.

Our goal is to reproduce real bugs and obtain, for each bug, a buggy and a fixed source-code version that differs by only the bug-fix patch. This section introduces the primary design of REDII and clarifies the details of the REDID and REDIT.

A. Dataset: REDID

1) Subject Selection: We selected 25 versions of 7 systems, including ActiveMQ [29], Cassandra [16], HDFS [30], HBase [18], Kafka [31], RocketMQ [32], and ZooKeeper [33]. They provide different system functionalities: HBase and Cassandra use different architectures to implement distributed databases; Kafka, RocketMQ, and ActiveMQ are message-queue systems for different application scenarios; ZooKeeper provides a distributed open-source application orchestration service. Each system has its unique architectural implementation and distributed error-correction mechanisms.

Given the long-term development and release of these systems, they utilize mature issue-tracking systems for software maintenance. These systems collect a wide range of issues, such as bug reports, enhancement requests, and questions. To support experimentation, the issue reports included in the dataset are required to meet the following criteria: (1) they are identified as the "bug" type; (2) these issues should be resolved and publicly accessible; (3) the version affected by the bug must be described in the JIRA issue; and (4) the fixed version should not contain the bug (*i.e.,* the subsequent tagged version). These criteria are essential because a bug could have been reported in, say v3.11.6, and also fixed in v3.11.6, and thus the bug would not be detected when using v3.11.6. These criteria ensure that the bugs we select are reproducible and always exist in a version of the artifact that can be accessed.

TABLE I: System Versions and Number of Bugs

System	Versions	Selected bugs
ActiveMQ	5.12.0, 5.14.0, 5.14.1, 5.15.0	10
Cassandra	2.2.16, 3.7, 3.11.3, 3.11.6	11
HDFS	2.7.0, 3.1.2, 3.2.0	9
HBase	1.4.0, 2.2.2, 2.4.9	7
Kafka	2.0.0, 2.8.0, 3.0.0, 3.3.1	9
RocketMQ	4.0.0, 4.5.0, 4.9.0	8
ZooKeeper	3.5.0, 3.5.3, 3.7.1, 3.6.0	9
Total		**63**

Manually reading and labeling all bug reports is a daunting and time-consuming task. So we first pulled the bug reports for each system on JIRA and randomly sampled 10% of them. For these bugs, we then manually screened them according to our criteria, leaving 10% as our dataset and finally obtaining a dataset of 63 bugs covering multiple versions and systems. We conducted a thorough examination of each bug report, ensuring the reliability of the test cases used to reproduce the bugs. The dataset includes a comprehensive test suite that is adaptable to various systems and versions, encompassing different types of bugs. This test suite has the capability to control bug injection and conduct automated testing, while also enabling reproducibility across diverse environments. We provide injection methods for these use cases in the manual.

2) Subject Setup: Reproduction is ensured through an accompanying test suite that includes at least one test case that exposes the bug — that is, the test case passes in the fixed version but fails in the buggy version [28]. When designing regression test suites for datasets, it is essential to first determine the specific version in which a particular bug was initially reported and fixed in its subsequent versions. To apply the patches effectively, we need to download the buggy version of the source-code package from the official website, which includes the project's source code and the required compilation tools. Different compilation tools offer distinct advantages and suitability for specific projects and scenarios. Due to variations in their build systems and configuration methods, diverse compilation commands may be required for different compilation tools.

In the dataset, each system version consists of a set of test suites. The test suite comprises the following components:

- **Source-Code Package:** Includes the source-code files of the distributed system, which can be compiled into an executable package.
- **Executable Package:** Enables users to effortlessly deploy and run distributed systems.

- **inject.c:** Simulates the patching process and offering the flexibility to decide whether to apply bug fixes based on the state of macro switches.
- **build.sh:** Performs tasks such as compiling and executing `inject.c`, building the system, and packaging it.
- **FaultSeeds.h:** Records the macro switches in `inject.c`.
- **path.txt:** Records the paths of the patch files.
- **Folders "buggy" and "fixed":** Stores source-code files necessary to switch between buggy and fixed versions.

The macro switches can be defined in the C code, and their values can be adjusted to control whether certain parts of the code are included or excluded during the compilation process. Using this feature, we introduce some macro switches in `inject.c`. Each of these macro switches corresponds to a bug that we selected in Section III-A1. By toggling these macros on or off, we can conditionally include or exclude specific parts of the code that represent the buggy or fixed behavior. This dynamic alteration facilitates toggling between the buggy- and fixed-code versions during compilation, thus enabling comparative testing and analysis. The purpose of providing these scripts is to simplify the control of single or multiple bug switches to support regression-testing research.

```
gcc -D'MACRO_1' -D'MACRO_2' (...) $cFile
    -o $exeFile
```

The test suite achieves the definition of macro switches for C files during the compilation process in `build.sh`. As shown below, it is possible to define multiple macros concurrently, which means we can inject multiple bugs for testing purposes. After the compilation and execution of `inject.c`, the files related to the patch are copied to the corresponding location in the source-code package. Subsequently, the script `build.sh` starts building the source code package. In our artifact, we present a use case illustrating our building process.

Over time, more effort is spent on revalidating the system's subsequent releases than on performing initial development testing [34]. The test suite that we provide can control the macro switches to choose whether to apply fixes, resulting in two distinct executable packages—the buggy and fixed version, supporting subsequent test-case construction and testing.

3) Test Case Construction: We construct a test case for each bug using REDIT. The process involved determining the triggering environment and event orders to create test cases for reproducing bugs. These may include injecting deterministic events, such as node crashes, network partitions and so on. The test cases primarily consist of two files: `ReditHelper.java` to define the deployment of the distributed environment and `SampleTest.java` to implement the testing process. Test cases consist of three kinds of operations:

- **Project API**: We use the API provided by the tested project to conduct specific operations. They are typically implemented through programming interfaces, which allow developers to interact with deployed projects in code.
- **System command**: To use the bash and some internal system calls, we must use system commands to help us build the scenario. These invocations are typically executed in a command-line interface, such as bash.
- **REDIT event**: We wrap the general operations used in distributed-system regression-testing as an event for developers. With the involvement of the event, developers can be dedicated to the key step for manually reproducing the bug's failure conditions and testing in deterministic manner.

The test cases implemented in the REDID enable us to reproduce the bugs and collect essential log data for further analysis. Furthermore, our test cases yield distinct results for the buggy version and the fixed version of the system. This provides compelling evidence of the effectiveness of REDID.

B. *Toolset:* REDIT

To deterministically reproduce the bugs in our dataset, we implement a framework, named REDIT, to assist test-case construction. Using REDIT, developers can obtain or construct test cases that can be deployed and executed with a predetermined execution scenario.

1) Deterministic Event Injection: Deterministic event injection refers to the process of introducing an event into a defined position within a sequence of execution, with the objective of maintaining a constant sequence of the designated event and its associated events throughout the ongoing operation of distributed systems [21]. We design REDIT with deterministic event injection as our primary goal of the design process, since a considerable number of failures of distributed systems can only be reproduced under deterministic constraints and event orders [21], [35]–[38]. These failures often involve consistency problems that require strict time constraints to be reliably reproduced. We treat both external and internal events uniformly as tasks and identify them as system methods. Additionally, we allow developers to give the order of execution of events. REDIT allows the definition of a set of internal events (*e.g.,* workload events, stack trace, block or unblock before/after a stack trace, and garbage collection), event injection, and a run sequence to accomplish this. In particular, a stack-trace event allows for the generation of time aspects at the statement level, enabling developers to perform deterministic event injection at a given runtime point. Further, block/unblock events allow developers to construct dependencies between events, ensuring sequential consistency during the execution of the events involved. Each of these events is named and can be bound together using the operator "$\star$" for sequential execution, "|" for parallel execution or no precedence, and parentheses "()" to form groups. A full list of event injections that REDIT currently supports can be found at [22]. Moreover, developers can customize their own required events by making custom extensions to REDIT.

Network Events. Network events are common in the deployment of distributed systems [39], and we implement many (including network partitioning, delay, and packet loss) for regression-testing scenario generation. This allows for the controlled simulation of real-world network conditions ,

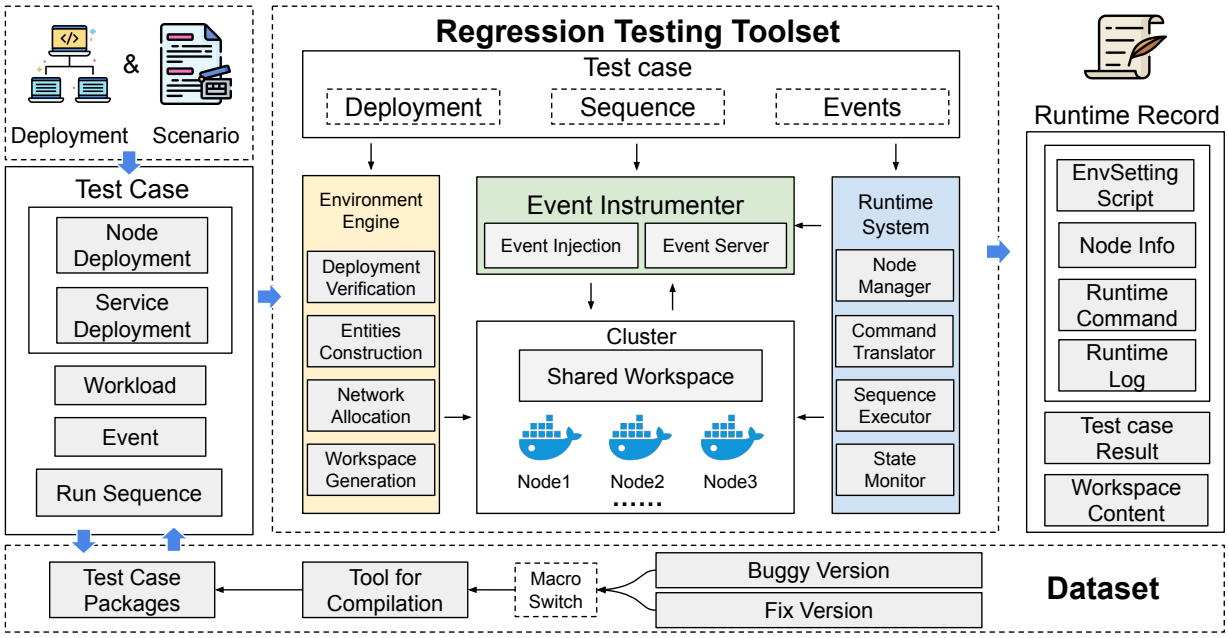

Fig. 1: The Architecture and Workflow of REDII.

which is useful for testing and evaluating the performance and resilience of applications and systems under various network scenarios. We manage data communication between containers by building firewall rules that allow/prohibit sending or receiving specific inter-container messages. The firewall rules can be dynamically adjusted during system operation with Bash scripts to trigger network partitioning at the point in time that we need. Simulating network latency and packet loss requires a different approach. We configure a flow-control queue on the NIC. For forwarded or received packets, we can set the packet-transmission speed and automatically insert delays during forwarding based on the configuration. After setting the packet-loss rate, the queue can randomly drop packets at a preset rate to simulate packet loss in the wild.

Node Events. Node events are also typical in distributed systems. Due to inherent instability of distributed systems, it is easy for nodes to crash and restart. We use the API provided by Docker to manage the state of nodes. Through the API, we can use Bash scripts within a Docker node to start and stop services or delete containers to end a node ultimately during the system running. We use node shutdown to simulate node crashes. Experimentally, node shutdown is largely sufficient for simulating crashes. In the scenario that a node needs to crash, we suspend work on other nodes and prioritize stopping the services running inside the container. After the service is shut down, we continue the work on the other nodes. Similarly, we can start the service inside the container if we need to restart a node. With the API provided by Docker, we can also add a new node while the cluster is running.

Clock Events. Since there is no absolutely uniform physical time-calculation method within a distributed system, the system is required to synchronize the clocks by some means. However, due to differences in local clock design, events at each node within the system can have slight deviations. We implemented clock-drift events with the `libfaketime` [40] library. We set a uniform virtual time when the *Runtime System* starts and use that virtual time to override the system-clock call service for each container. In this way, we shield the containers from looking up the real system time. When we modify the virtual time of some nodes, we can make the node's system-time invocation service get the time that we specify to simulate clock drift in physical clocks. With the clock-drift event, we provide the possibility to reproduce some exceptions within the distributed system due to the wrong timing of events.

2) Environment Engine: We use an *Environment Engine* to help users implement system deployments. We implement "configure-as-code" functionality that allows developers to deploy systems layer-by-layer with a hierarchical builder model. This configuration code is translated into Docker container generation and configurations.

Environment Engine consists of four main steps: (1) *Deployment Verification*, (2) *Entities Construction*, (3) *Network Allocation*, and (4) *Workspace Generation*. REDIT first determines whether a system build-script can be generated from the configuration, based on constraints (*e.g.,* address legality, duplicate nodes, and duplicate services). We assist in configuring the properties of the system by calling a series of methods. For configured elements, we create and add child elements (*e.g.,* nodes) entities via the "*with*" method of deployment. We call methods like "*dockerFileAddress()*" based on the child element entity to configure the properties of the child element. Next, we call back the deployment entity with an "*and()*" method. Through this process, REDIT allows the user to construct the system to be deployed in a hierarchical manner. Configurations consist of all the components needed for the system to be deployed on the nodes.

Deployment Verification iteratively executes the items in the configuration. Each time the configuration is executed, it is verified whether the configuration meets the requirements of the specific part, the pre-defined configuration needs of REDIT, and the constraints of the system itself. If it completes successfully, the configuration is proven to be error-free and the system can continue to be built. After verification, REDIT starts creating new containers based on the build-script content. The container base environment is generated based on the initial Dockerfile, and REDIT extracts the service packaging files to the corresponding Docker containers according to the services required by the system and initializes the services using the related Bash scripts. We use the above approach to deploy the system for each set of nodes and services. Then, REDIT assigns each container a virtual network address in order and writes these network addresses to the routing table to set up the cluster network for further use by *Runtime System*. REDIT continually generates new network addresses by accumulating based on the initial IP address and shares a complete table of container network addresses via *Runtime System*. While each node actually holds address information for other nodes within the distributed system, whether or not it can be accessed is still dependent on the setting of firewall rules. Finally, REDIT creates shared folders to synchronize some additional files like global scripts and shared data files and sets the working path for each service. This allows each service to independently consider its own run path, decoupling the scripting of commands within the service from the address where the service actually runs and increasing the uniformity of scripting. REDIT stores all the initialization-variable information in its own deployment object after deploying the system and passes this object to the *Runtime System*.

3) Event Instrumenter: Event Instrumenter is one of the extension points of our tool. Given the deployment definition, this component can instrument the nodes' binaries. We have two main parts here: *Event Injection* and *Event Server*. With this component, we implement the injection of deterministic events and the deterministic execution of event sequences. The *Event Injection* part reads the run sequence from the test case file. First, our tool generates a series of event injection points based on the run sequence and then injects the events into the breakpoint locations based on the language-specific breakpoint tools. With ASPECTJ [41], we can implement event-breakpoint injection in languages running on JVM. Based on the API provided by the *Runtime System*, our tool can perform a series of event injections, including network events, node events, and clock events. REDIT allows developers to manually construct conditional event constraints on event-injection points. After the user configures the event sequence using mechanisms such as locks, our tool automatically generates the corresponding conditional constraints for *Event Server*, including allowing blocking, enforcing orders, garbage collection, and so on.

The *Event Server* is started when a running sequence is included in the deployment. We implement the *Event Server* using JETTY [42] and allow this server to communicate with all nodes. After getting the event sequence, it executes the sequence based on constraints and monitors the execution of sequence. The *Event Server* knows blocking conditions, dependencies, and the current status of all events in the running sequence. It cooperates with *Sequence Executor* to dynamically adjust the running state of the event sequence.

4) Runtime System: Runtime System is another extension in REDIT. Given the deployment definition and the created nodes' workspaces, this component deploys the system and provides proper interfaces to manipulate the deployed environment. The component includes four parts: *Node Manager*, *Command Translator*, *Sequence Executor*, and *State Monitor*.

Docker enables REDIT to monitor node status, start/stop nodes, and add new nodes. Docker API is utilized to manage node state, generate node events, and execute script files within nodes. The developer can also detect and record the container state at each node during the event sequence running, which can be used to record the status of the entire run of the regression test and to analyze any errors that may have occurred. With these features, REDIT implements a *Node Manager*, which packages Docker's functionalities as APIs for *Event Instrumenter* and other parts in *Runtime System*.

In order to eliminate the need for users to pay attention to the underlying implementation, we try to avoid script commands executed in the nodes. *Command Translator* reads the variable parameters in the deployment and searches according to their identifiers. After getting the environment parameters, *Command Translator* rewrites the user's commands into a script and writes it to the working directory. This allows developers to focus only on the alignment of event sequences and the design of event contents, from which REDIT's ability to run universal tests across platforms is derived. Next, *Command Translator* corresponds the ID of the script to the function to be executed and passes it to *Sequence Executor*.

Sequence Executor accepts the operation methods provided by the *Node Manager* and *Command Translator*. According to interfaces provided by the *Event Server*, *Sequence Executor* calls the corresponding method to execute the sequence on the deployed system. After the event sequence is completed, *Sequence Executor* packages the files and logs of the deployed system and stores them in the host's working directory.

State Monitor detects the state of the framework and the deployment system. *State Monitor* organizes the logs within the deployment system into a dedicated log directory to facilitate the export of the system's working logs at the end of the test. At the same time, *State Monitor* detects the operation of REDIT and writes REDIT's operating environment to the local logs. In the event that REDIT crashes abnormally, *State Monitor* records the current status in the logs, to allow for easy discovery of the cause and location of errors for developers.

IV. CASE STUDIES AND ANALYSIS

To demonstrate the effectiveness and efficiency of REDIT in deterministic bug regression testing framework, we provide two research questions to clarify our research and study goals. We select three test cases from REDID for a detailed description and provide performance analysis with REDIT.

A. Research Questions

RQ 1: *Can* REDII *be used to* ***effectively*** *perform regression testing?* The most critical task for a regression-testing system is to allow failure reproduction and testing. In order to evaluate the effectiveness of the framework, we randomly selected multiple issues in multiple systems from issue-tracking systems, like JIRA, for reproduction as described in Section III. To demonstrate the effectiveness of our study, we use the case-study method to select 3 typical cases for introduction, analysis, and demonstration of our practice in REDIT.

RQ 2: *To what degree can* REDII *improve the* ***efficiency*** *in regression testing?* Efficiency in regression testing was evaluated based on several key factors including the time required to execute tests, utilize resources, and ease efforts. Due to the lack of corresponding JUnit tests for most of the tests, we were only able to compare the performance of a few test cases, so we turned to the current test conditions to find out if the test performance was acceptable. The performance metrics show that the results of the performance analyses, such as test time and resource consumption, are within acceptable limits while ensuring the effectiveness of the reproduction.

B. Answering RQ1: Case Studies

Case 1. CASSANDRA-14365 [43]: Commit log replay failure for static columns with collections in clustering keys.

The bug in CASSANDRA-14365 is that a restarted cluster node exits when fails to process the commit log during initialization for the replay of tables with a same name but incompatible schemas. Due to an unexpected error in deserialization mutation, the verification cannot be performed, causing the failure to join the cluster. This bug occurs in v2.2.16 and earlier. In this case, two nodes are necessary to trigger the bug, and *cassandra.yaml* is the necessary configuration file for the cluster, so we customize the file and mount it.

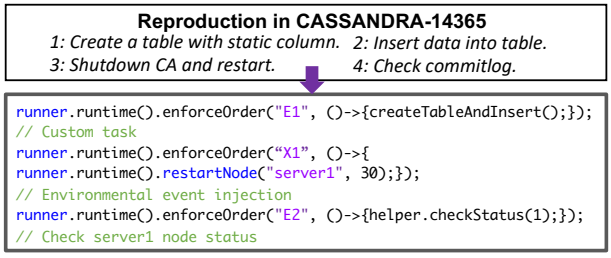

Fig. 2: From Reproducing Steps to REDIT Test Case Event Order with "restartNode"

A snippet of code that highlights the critical event injection is depicted in Figure 2. The test case attempts to cause the cluster to reprocess flawed commit logs by forcing the injection of table flaws and node crashes. Event E1 is defined as the precondition for triggering the defect, which is implemented by the *createTableAndInsert* method. Based on the script provided in the bug report, *createTableAndInsert* method creates a table with a static column on the node and inserts data into it. This event lets us know when to failover. After the workload completes, in order to simulate the bug implementation of static columns with collections, it creates a session with the cluster through the API, then creates a table structure of type `frozen<map<text, text>>` and inserts a piece of test data. The Cassandra cluster writes to the commit log as soon as it receives the writing task. At this time, REDIT's bug injection method *restartNode* is called to inject the node-crash bug and to force the restart of the `server1` container. Compared to the test cases that are provided with CASSANDRA-14365, REDIT can construct bug scenarios directly based on the steps to reproduce in the bug report, which clearly enables us to build regression tests in a more efficient way.

Case 2: ZOOKEEPER-2355 [44]: Ephemeral node is never deleted if follower fails while reading the proposal packet.

A critical bug was raised in ZOOKEEPER-2355 that ZK's ephemeral node is never deleted if a follower fails while reading the proposal packet. If the client connects to any server at this time and creates an ephemeral node with the same path, the creation fails because the old ephemeral node is not deleted normally, which violates the reliability principle.

```
1  runner.runtime().enforceOrder("E1", ()-> {
2     // Create ephemeral node
3     zkClient1.create(nodePath, "1".getBytes(),
           ZooDefs.Ids.OPEN_ACL_UNSAFE,
          CreateMode.EPHEMERAL);});
4  runner.runtime().enforceOrder("X1", ()-> {
5     // Inject network partition
6     NetPart netPart = NetPart.partitions("
          follower1", otherNodes).build();
7     runner.runtime().networkPartition(netPart)
          ;});
8  runner.runtime().enforceOrder("E2", ()-> {
9     // Close the session to delete the
          ephemeral node
10    zkClient1.close(); });
11 runner.runtime().enforceOrder("X2", ()-> {
12    // Remove network partition
13    runner.runtime().removeNetworkPartition(
          netPart);});
14 runner.runtime().enforceOrder("E3", ()-> {
15    // Check the ephemeral node status
16    Stat stat = zkClient2.exists(nodePath,
          false);});
17 runner.runtime().enforceOrder("E4", ()-> {
18    // Create node with another session
19    zkClient2.create(nodePath, "2".getBytes(),
           ZooDefs.Ids.OPEN_ACL_UNSAFE,
          CreateMode.EPHEMERAL);});
```

Listing 1: REDIT Test Snippet for Network Partition Injection

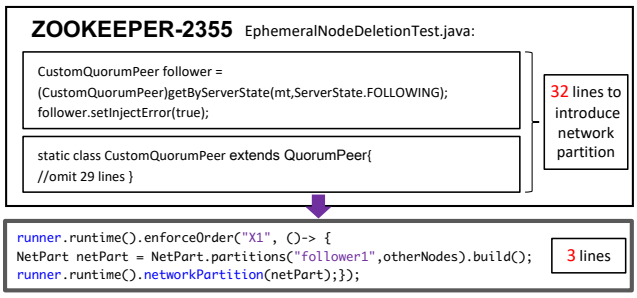

Fig. 3: Code Reduction with REDIT for Network Partition Injection

Listing 1 shows 4 general events and 2 network events, and

the order of them is determined. After the ZooKeeper cluster is started, we create a client named zkClient1 to connect to the cluster for the subsequent creation of an ephemeral node. Event E1 is defined as zkClient1 creates an ephemeral node "1" to use a fully developed ACL and allow every client to read/write to the ZNode. To simulate a follower failing to read the proposal packet, we impose a network partition on one of the followers to isolate it from the others (X1) before closing the zkClient1. The life cycle of the ephemeral node is bound to the client session. If the client session is invalid, the node should be automatically cleared. So we created a new session to verify the existence of the ephemeral node (E2) after removing the network partition (X2). Further, we define E3 as creating ephemeral nodes on the same path for a new client. The old ephemeral node is not removed, the client's request is rejected with a `NodeExistsException`. According to the bug description provided by the publisher, we confirmed that this bug can be reproduced stably using v3.4.8 based on REDIT. Figure 3 demonstrates the difference between the test case implemented with REDIT and the one provided in Zookeeper.

Case 3: ZOOKEEPER-1366 [45]: ZooKeeper should be tolerant of clock adjustments. In distributed systems, the clocks of nodes can drift. This requires a distributed-systems project that can tolerate or fix the clock drift that exists. And ZOOKEEPER-1366 states that there are incidents where ZooKeeper uses absolute time. If severe clock drift exists, they may cause sessions to expire immediately. The test provided in the bug report requires external control of the current time state using `[date -s "+1hour"]`. This is tedious and cumbersome. The external command was not guaranteed to consistently trigger the error, and subsequent developers were unable to understand the hidden clock check from the test code, which in fact led to recurring related issues for years after the bug was fixed. We rewrote this case using REDIT and constructed the scenarios required for this bug by simply creating time drifts for nodes with a single line of code in Listing 2. We tested this on version 3.5.0-alpha.

```
1  runner.runtime().enforceOrder("E1", () -> {
2     ZKClientConnect();// create a zk client
3     ZKCreateNode(ZPATH_1,config_2); });//
          create a new node
4  runner.runtime().enforceOrder("X1", () -> {
5     runner.runtime().clockDrift("server",
          50000);});
6     // Apply 50s clock drift
7  runner.runtime().enforceOrder("E2", () ->{
8     ZKCreateNode(ZPATH_2,config_2);}); //
          create a new node
```

Listing 2: REDIT Test Snippet for Clock Drift Injection

We first define three Docker containers and start the ZooKeeper cluster. Once the cluster is running, it automatically starts the leader election. Then we create a zk client and a zk node with this client to simulate the new zk node operation under normal circumstances. After that, we apply clock drift on all zk servers using REDIT. The clock drift is larger than the session timeout (*e.g.,* 50s). When we try to create a zk node again using the same client, it should fail with `SessionExpiredException`.

In comparison to the conventional approach of relying on external bash commands to manage the clock event in ZOOKEEPER-1366, REDIT introduces a more explicit and robust solution, eliminating unpredictability and uncertainty in distributed system testing.

TABLE II: Performance Indicators of Regression Testing

Bug ID	Time	PIDS	CPU%	MEM USAGE	MEM%
ActiveMQ	22.92s	62.98	19.51%	228.00MiB	0.73%
Cassandra	31.48s	90.92	42.37%	4665.98MiB	14.85%
HBase	146.96s	138.97	8.45%	375.95MiB	1.20%
HDFS	77.20s	89.34	10.80%	240.97MiB	0.77%
Kafka	41.10s	152.42	14.79%	337.78MiB	1.08%
RocketMQ	36.58s	112.99	21.45%	960.82MiB	3.06%
ZooKeeper	55.06s	54.56	7.18%	70.92MiB	0.23%

Summary of Case Studies: Our case studies evaluate bug reproduction and testing across three distributed system types, highlighting REDIT's robust regression testing capabilities and affirmatively answering RQ1. By utilizing deterministic event order, we implemented regression tests for node event regression, network partition complexities, and advanced functionalities beyond traditional unit tests.

C. Answering RQ2: Performance Analysis

We evaluate the practicability of 7 open-source distributed systems in the REDIT, whose testing methods are quite different from those of traditional distributed systems. We choose system bugs whose steps to reproduce contain at least one distributed environment failure (*e.g.,* node crash, network partition). Compared to traditional testing frameworks such as JUnit [46], which can only use test tools written by system developers and cannot implement such custom injection of bugs, REDIT shows great cross-platform ability and generality.

The evaluation of performance metrics in REDIT is detailed in Table II, which presents the outcomes of various scenarios executed under REDIT. To ensure data reliability and minimize uncertainties, we conducted 3 iterations of dataset testing and calculated the average number of performance metrics. The table's second column displays the time required to replicate the bugs, while the third column represents the average number of concurrent Process IDs (PIDs) across all containers, suggesting low time overhead and concurrency in REDIT's code. The fourth column reports the CPU utilization for all containers during testing. Notably, containerized environments, especially those involving Cassandra, HDFS, and HBase clusters, exhibit elevated CPU usage due to significant I/O processes, yet this remains within acceptable limits for standard computing equipment. The final two columns of the table indicate memory usage during testing. Operating on a system with 32GB of RAM (31.14GB available to Docker), the average memory consumption across tests was below 15%, signifying a minimal impact. Overall, REDIT demonstrates

robust performance in multi-container settings, excelling in both time efficiency and resource management.

Summary of Performance Analysis: We conduct a quantitative analysis of time, and system resource consumption. REDIT's operation across all test cases does not impose a significant system load. This demonstrates its capability to perform comprehensive testing without compromising system performance, affirmatively answering RQ2.

V. DISCUSSION

A. Usage Scenarios

REDII is well-suited for analyzing deterministic-related bugs and reproducing failures. The main challenge in writing REDIT test cases was not the REDIT framework itself but understanding distributed systems and the runtime conditions leading to failures. Understanding distributed-system source code required significant manual effort but would not be an issue for the original developers. Runtime conditions were already diagnosed and documented in bug-tracking systems when fixes were implemented. Lastly, the REDIT framework consists of simple library classes and methods using the builder pattern, documented in the artifact repository.

Meanwhile, it is useful for software-engineering researchers to bring many of the advanced analyses and techniques that we have created for monolithic systems to the field of distributed system software engineering. For **developers** and **researchers**, we imagine usage scenarios for REDII, such as:

- Enabling the creation of test suites for future, new distributed systems software codebases using a generic, cross-project testing framework without the need for developers to recreate their own custom test scaffolding that cannot be used for other systems;
- Allowing the manual creation of new test cases that reproduce the deterministic failure conditions (including event timings, network failures, etc.) alongside the bug fixes, in order to prevent future regressions;
- Assisting in rewriting and resolving existing flaky tests in current test suites;
- Creating novel bug detection, localization, testing, and repair techniques that are targeted at distributed computation and the unique types of bugs that they suffer from; and
- Conducting empirical studies on test-case prioritization for distributed systems.

Such a list of potential uses of REDII is clearly not exhaustive, but we hope that it can inspire the breadth and diversity of potential innovation.

B. Scope of Applicable Distributed Systems

Currently, our implementation is focused on platforms written in Java or those that compile to Java bytecode, *i.e.,* systems that can run on the Java Virtual Machine , and the dataset REDID contains the bugs from such systems. This implementation scoping is partially due to REDIT's use of AspectJ [41] to flexibly instrument and monitor the execution process. In distributed systems, a key challenge is controlling environmental conditions to reproduce and debug hard-to-capture bugs, such as those related to event ordering, network partitioning, or clock drift. Consequently, developing reliable testing environments that accurately mimic real-world conditions is crucial for resolving these issues and ensuring the robustness of distributed systems. By leveraging AspectJ, we can insert cross-cutting concerns (*e.g.,* logging, performance monitoring, and error handling) without modifying the core business logic. A full list of event injections that REDIT supports can be found at [22]. Moreover, developers can customize their own required events by making custom extensions to REDIT. In the future, we plan to expand the infrastructure to support a wider range of platforms and programming languages such as C/C++, enhancing its flexibility and adaptability.

C. Key Strengths of REDII

To the best of our knowledge, REDII contains the first dataset based on real bugs to support deterministic bug testing and the first deterministic regression-testing framework for distributed systems. REDIT is a lightweight and generalizable (*i.e.,* can be used by multiple projects) test framework for developers to write tests for validating the correctness of distributed systems under specific-ordered events. After REDIT is configured, developers can construct an environment deployment based on a template for the target system and create appropriate workloads, perturbations, and event sequences. In distributed systems, many failures involving multiple events only occur if the events happen in a specific order [21], [35]–[38]. REDIT implements common environmental events that affect distributed systems, enabling testers to inject these events in a specific order using REDIT APIs. REDIT supports deterministic event injection, allowing the efficient reproduction of distributed failures, which is intended to assist detection and prevention of future regressions. Our approach provides a general, flexible, and extensible approach to distributed-system testing. Moreover, our dataset provides real bugs and replayable environmental conditions needed to trigger the bugs to demonstrate test failure, each of which was reproduced from real bug reports from real distributed systems. This dataset can be used by researchers in future efforts in the field of distributed-system testing.

Compared to thread-based testing, which simulates network events by suspending threads [5], REDIT can create distributed system nodes in different Docker instances to simulate deployment. REDIT provides diverse events and simplifies the process of writing test cases by focusing on regression tests rather than designing thread order, locks, and events. This feature ensures correct behavior in the evolution of distributed systems and can integrate into the software-development process, such as through continuous integration, thus improving test efficiency. It enables complex multi-node parallel tests on a single machine and simplifies the transition to real distributed systems. Compared to thread-based testing methods [17], [18], REDIT employs Docker to abstract underlying details, which allows the system to perform cross-node communication using its native protocols (*e.g.,* HTTP requests) and also allows for per-test-specific environmental controls.

D. Threats to Validity

The failures in case studies cannot encompass all possible scenarios and use cases of distributed system environments. To alleviate this threat, we select widely used distributed systems for different usages across various applications to improve the coverage of REDIT. Additionally, we have designed REDIT to be adaptive for future enhancements and extensions. We design three distinct kinds of operations, covering necessary elements for effective test-case creation. As shown in Section IV-B, test cases can be constructed by these operations, providing effective and efficient regression testing. Since the verification of clock bugs mostly relies on direct checks of system variables and lacks environmental simulation, we can only use clock events in a few cases. Since the clock events replace external bash commands, we calculate the code reduction in terms of the number of command lines. Another threat to validity comes from the participant selection. We employ four participants to write tests and conduct the study and analysis. Developers may write the test code in different ways, resulting in various performance and code sizes. To relax this threat, all participants are students without much development experience. The results reflect the performance of REDIT handled by newcomers. In practice, professional developers should have more expertise regarding the target distributed systems, thus they may be likely to gain even higher efficiency improvements from REDII when developing tests.

VI. RELATED WORK

Regression Testing Frameworks. Some distributed systems, such as Cassandra, Hadoop, Kafka and HBase [16]–[18], [31] have dtest, Hadoop MiniCluster, Kafka Cluster and HBase MiniCluster for regression testing. While they use pseudo-distributed designs, avoid the complexity of system deployment in a real cluster and assert certain states and behaviors for the developers, each of them is tightly coupled to their target system. They are focused on creating specialized solutions rather than investing in a more generalized solution that could be utilized by other systems. There are some regression testing framework efforts for distributed systems [19], [20], but they either don't provide adequate dataset support or don't implement the convenient event abstraction in REDIT, and are more oriented towards fault detection. REDII addresses these issues by providing a unified platform and incorporates support for deterministic event injection, allowing for more realistic simulations of real-world scenarios.

Random Failure Injection for Distributed Systems. Recently some tools have focused on providing bug injection to trigger potential problems in the system, *i.e.,* to simulate an environmental state with code injected into the system. These tools can be classified into external and internal bug injection, depending on whether the system's code is modified. However, some tools inject only a single bug, such as network partition (*e.g.,* [35]), node crashes (*e.g.,* [47]), etc., which prevents them from exploring other errors. Some tools provide random bug injection, such as Namazu [48], JEPSEN [26] or MALLORY [49]. They randomly inject various faults like node faults and network delay to simulate unstable system states and perform large-scale exploration of possible error states in the system. REDIT has no intention to compete with these tools. It works as a complementary tool for these random injection tools as they could provide event sequences that could be reproduced as a deterministic test case. REDIT allows developers to create a test case that prevents regressions in a deterministic manner with minimal effort, significantly improving the effectiveness and efficiency of testing in the evolution of distributed software systems.

Model-based Checker for Distributed Systems. Some frameworks (*e.g.,* [4], [15], [50]–[52]) use a model checker to determine whether the distributed system satisfies the constraints given by itself. These works intercept non-deterministic distributed events, including node crashes and order permutation. Through methods like heuristic algorithms, model checkers can find problems that are difficult to detect in wild. Some model checkers (*e.g.,* [13], [14]) implement an optimized policy design that allows them to check the distributed system in less time. However, the problem of time-consuming heuristic rules has not been fundamentally resolved. For regression testing in real-world development, developers prefer to prioritize checking for error scenarios that have already occurred. Developers want the system to be guaranteed correct in these scenarios already known, but the model checker is more focused on searching for all state spaces. REDIT, on the other hand, is more devoted to regression testing and can help developers reach this goal much more quickly.

VII. CONCLUSION

In this paper, we propose our infrastructure REDII to support reproducing deterministic distributed systems failures, including a rich bug dataset REDID and REDIT, a deterministic and generic failure regression-testing framework for distributed systems. We evaluate REDIT with our dataset collected from JIRA on seven systems. Our case studies show that REDIT enables developers to deterministically reproduce runtime failures for distributed-system bugs, which are triggered by potentially non-deterministic environmental events or specific event sequences. The performance analysis shows that our framework is also efficient for real-world distributed system regression testing. In the future, we plan to focus on automatic consistency violation detection frameworks and a more automated approach for dataset generation, access, and evaluation based on REDII to help developers detect unknown bugs and translate them into regression-testing scenarios.

ACKNOWLEDGMENT

We would like to thank Armin Balalaie for his work in developing the early prototypes of the testing infrastructure [53]. This project was partially funded by the National Natural Science Foundation of China under Grant No. 62372225 and No. 62272220.

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, p. 67–120, Mar. 2012. [Online]. Available: https://doi.org/10.1002/stv.430

[2] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–16. [Online]. Available: https://doi.org/10.1145/2987550.2987583

[3] P. Alvaro and S. Tymon, "Abstracting the geniuses away from failure testing: Ordinary users need tools that automate the selection of custom-tailored faults to inject." *Queue*, vol. 15, no. 5, p. 29–53, oct 2017. [Online]. Available: https://doi.org/10.1145/3155112.3155114

[4] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, Mar. 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

[5] G. Milka and K. Rzadca, "Dfuntest: A testing framework for distributed applications," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer International Publishing, 2018, pp. 395–405.

[6] B. White, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. Boston, MA: USENIX Association, Dec. 2002. [Online]. Available: https://www.usenix.org/conference/osdi-02/integrated-experimental-environment-distributed-systems-and-networks

[7] P. Joshi, H. S. Gunawi, and K. Sen, "Prefail: a programmable tool for multiple-failure injection," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 171–188. [Online]. Available: https://doi.org/10.1145/2048066.2048082

[8] B. K. Ozkan, R. Majumdar, and S. Oraee, "Trace aware random testing for distributed systems," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360606

[9] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, and L. Hochstein, "Automating failure testing research at internet scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 17–28. [Online]. Available: https://doi.org/10.1145/2987550.2987555

[10] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 331–346. [Online]. Available: https://doi.org/10.1145/2723372.2723711

[11] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 677–691, apr 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037735

[12] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "Fcatch: Automatically detecting time-of-fault bugs in cloud systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 419–431. [Online]. Available: https://doi.org/10.1145/3173162.3177161

[13] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 399–414. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

[14] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi, "Flymc: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303986

[15] J. Simsa, R. Bryant, and G. Gibson, "dBug: Systematic evaluation of distributed systems," in *5th International Workshop on Systems Software Verification (SSV 10)*. Vancouver, BC: USENIX Association, Oct. 2010. [Online]. Available: https://www.usenix.org/conference/ssv10/dbug-systematic-evaluation-distributed-systems

[16] C. D. Team, "Cassandra," 2024. [Online]. Available: https://github.com/apache/cassandra

[17] H. D. Team, "Hadoop," 2024. [Online]. Available: https://github.com/apache/hadoop

[18] H.-B. D. Team, "HBase," 2024. [Online]. Available: https://github.com/apache/hbase

[19] E. Gabrielova, "End-to-end regression testing for distributed systems," in *Proceedings of the 18th Doctoral Symposium of the 18th International Middleware Conference*, ser. Middleware '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 9–12. [Online]. Available: https://doi.org/10.1145/3152688.3152692

[20] M. Babaei and J. Dingel, "Efficient replay-based regression testing for distributed reactive systems in the context of model-driven development," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2021, pp. 89–100.

[21] K. Li, P. Joshi, A. Gupta, and M. K. Ganai, "Reprolite: A lightweight tool to quickly reproduce hard system bugs," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–13. [Online]. Available: https://doi.org/10.1145/2670979.2671004

[22] RediI, "REDII artifact: Dataset, test framework, and documentation," https://anonymous.4open.science/r/RediI-0EB1/, 2025.

[23] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 19–33. [Online]. Available: https://doi.org/10.1145/3132747.3132768

[24] riptano, "ccm," 2024. [Online]. Available: https://github.com/riptano/ccm

[25] Elasticsearch, "Elasticsearch testing framework," https://github.com/elastic/elasticsearch/blob/main/test/framework/src/main/java/org/elasticsearch/test/InternalTestCluster.java, 2022.

[26] Jepsen-IO, "Jepsen," https://github.com/jepsen-io/jepsen, 2022.

[27] H. Chen, W. Dou, D. Wang, and F. Qin, "CoFI: consistency-guided fault injection for cloud systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 536–547. [Online]. Available: https://doi.org/10.1145/3324884.3416548

[28] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[29] A. D. Team, "ActiveMQ," 2024. [Online]. Available: https://github.com/apache/activemq

[30] H.-H. D. Team, "Hadoop-HDFS," 2018. [Online]. Available: https://github.com/apache/hadoop-hdfs

[31] K. D. Team, "Kafka," 2024. [Online]. Available: https://github.com/apache/kafka

[32] R. D. Team, "RocketMQ," 2024. [Online]. Available: https://github.com/apache/rocketmq

[33] Z. D. Team, "ZooKeeper," 2024. [Online]. Available: https://github.com/apache/zookeeper

[34] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, no. 4, p. 405–435, oct 2005. [Online]. Available: https://doi.org/10.1007/s10664-005-3861-2

[35] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 51–68.

[Online]. Available: https://www.usenix.org/conference/osdi18/presentation/alquraan

[36] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2020, p. 339–351. [Online]. Available: https://doi.org/10.1109/ASE.2019.00040

[37] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 249–265. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[38] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, "Setsudō: perturbation-based testing framework for scalable distributed systems," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, ser. TRIOS '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2524211.2524217

[39] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 73–100, 2011.

[40] W. Hommel, "libfaketime," https://github.com/wolfcw/libfaketime, 2022.

[41] Eclipse-AspectJ, "Aspectj," 2022. [Online]. Available: https://github.com/eclipse/org.aspectj

[42] Eclipse-Jetty, "Jetty," 2022. [Online]. Available: https://github.com/eclipse/jetty.project

[43] CASSANDRA-14365, "Cassandra jira issue 14365," https://issues.apache.org/jira/browse/CASSANDRA-14365, 2015.

[44] ZK-2355, "Zookeeper jira issue 2355," https://issues.apache.org/jira/browse/ZOOKEEPER-2355, 2016.

[45] ZK-1366, "Zookeeper jira issue 1366," https://issues.apache.org/jira/browse/ZOOKEEPER-1366, 2015.

[46] JUnit-Team, "Junit5," https://github.com/junit-team/junit5, 2022.

[47] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 114–130. [Online]. Available: https://doi.org/10.1145/3341301.3359645

[48] Osrg, "Namazu," https://github.com/osrg/namazu, 2016.

[49] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox fuzzing of distributed systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1615–1629. [Online]. Available: https://doi.org/10.1145/3576915.3623097

[50] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: Transparent model checking of unmodified distributed systems," in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. Boston, MA: USENIX Association, Apr. 2009. [Online]. Available: https://www.usenix.org/conference/nsdi-09/modist-transparent-model-checking-unmodified-distributed-systems

[51] V. Anand, "Dara: hybrid model checking of distributed systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 977–979. [Online]. Available: https://doi.org/10.1145/3236024.3275438

[52] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, Apr. 2007. [Online]. Available: https://www.usenix.org/conference/nsdi-07/life-death-and-critical-transition-finding-liveness-bugs-systems-code

[53] A. Balalaie and J. A. Jones, "Towards a library for deterministic failure testing of distributed systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 486. [Online]. Available: https://doi.org/10.1145/3357223.3366026