Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker

Yinfang Chen, Xudong Sun, Suman Nath[†], Ze Yang, Tianyin Xu

University of Illinois at Urbana-Champaign [†]Microsoft Research

Abstract

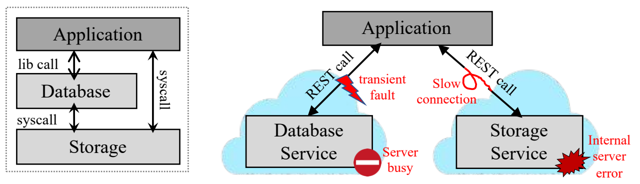
Modern applications have been emerging towards a cloud-based programming model where applications depend on cloud services for various functionalities. Such "cloud native" practice greatly simplifies application deployment and realizes cloud benefits (e.g., availability). Meanwhile, it imposes emerging reliability challenges for addressing fault models of the opaque cloud and less predictable Internet connections.

In this paper, we discuss these reliability challenges. We develop a taxonomy of bugs that render cloud-backed applications vulnerable to common transient faults. We show that (mis)handling transient error(s) of even one REST call interaction can adversely affect application correctness.

We take a first step to address the challenges by building a "push-button" reliability testing tool named Rainmaker, as a basic SDK utility for any cloud-backed application. Rainmaker helps developers anticipate the myriad of errors under the cloud-based fault model, without a need to write new policies, oracles, or test cases. Rainmaker directly works with existing test suites and is a plug-and-play tool for existing test environments. Rainmaker injects faults in the interactions between the application and cloud services. It does so at the REST layer, and thus is transparent to applications under test. More importantly, it encodes automatic fault injection policies to cover the various taxonomized bug patterns, and automatic oracles that embrace existing in-house software tests. To date, Rainmaker has detected 73 bugs (55 confirmed and 51 fixed) in 11 popular cloud-backed applications.

1 Introduction

Modern applications have been emerging towards a cloud-based programming model where applications depend on cloud services for various functionalities. Such "cloud native" practice greatly simplifies application development and deployment, and realizes cloud benefits (e.g., scalability, availability, and cost efficiency). Today, all major cloud providers offer various cloud services to support cloud-based programming, e.g., storage, database, and machine learning [5, 10, 14]. These cloud services have been increasingly adopted, e.g., the .NET SDK of Azure Storage services has tens of thousands of daily downloads [70]. We term the applications that rely on cloud services *cloud-backed applications*.



(a) Traditional applications (b) Cloud-backed applications

Figure 1: Fault domains of (a) traditional applications and (b) cloud-backed applications (the subjects of this paper).

Cloud-backed applications interact with one or more cloud services, usually through REST APIs over HTTP/HTTPS. To ease programming, cloud providers typically offer SDKs on top of the REST APIs to support applications written in different programming languages. For example, AWS provides SDKs in 12 languages, such as .NET, Java, Python, and C++.

Despite the attractive benefits, cloud-based programming imposes emerging reliability challenges introduced by the fault models of opaque cloud backends and less predictable connections between the application and cloud services. Figure 1 compares the fault model of cloud-backed applications with traditional applications backed by local services. Unlike traditional applications that have simple, shared fault domains as the system services with well-specified APIs (e.g., POSIX), the fault domains of cloud-backed applications are more heterogenous, unpredictable, and opaque. It is reported that cloud-backed applications commonly experience transient errors and network delays [22, 42, 81, 88].

In this paper, we unravel the reliability challenges faced by cloud-backed applications. We show that there is a lack of standards and consistencies of existing cloud services on what errors are communicated by cloud service APIs, and how SDKs handle the errors. As a consequence, it is challenging for application developers to anticipate and correctly handle myriad faults that could occur during the application's interaction with the cloud services, resulting in critical bugs. For example, many SDKs employ automatic retries to handle transient errors; however, retries on non-idempotent APIs, if not done correctly, could result in elusive behavior, such as silent semantic violations and unhandled exceptions (see §3).

Contributions. We take a first step to address the emerging reliability challenges by building a "push-button" reliability testing tool named Rainmaker, as a basic SDK utility for any cloud-backed application. Rainmaker helps developers *easily* and *systematically* test their applications' correctness, in the face of various errors under the cloud-based fault model. Rainmaker does not need developers to write policies, oracles, or test cases. It directly works with existing test suites and is a plug-and-play tool for existing test environments. Rainmaker is generic to any type of cloud-backed applications.

Designing Rainmaker is challenging. Despite the rich literature on fault injection techniques and error-handling analysis (see §7), we find that no technique can support a push-button solution for cloud-backed applications. Many existing tools provide only the basic randomized fault injection policies and basic crash oracles [6, 37, 47] that can miss critical bugs, and application-specific techniques are not widely applicable (e.g., checking data consistency for databases [16, 52] and file systems [20, 69]). Techniques that address program faults (exceptions and errno) and component faults (e.g., node crashes) are too coarse-grained to capture the nuances of complex interactions between the application and cloud services.

Rainmaker puts its focus on *transient* errors faced by cloud-backed applications. Basically, Rainmaker injects transient faults (e.g., temporary service unavailability and request timeouts) by intercepting outbound REST API calls from the application to the cloud service at the HTTP layer. HTTP-layer interception makes it easy to capture fine-grained interactions (including those triggered by SDK-level retries) and is transparent to applications under test. Hence, Rainmaker requires no modification of application source code and can be directly applied to an existing test environment.

A key component of Rainmaker is its *automatic* fault injection policies that define 1) *what* faults to inject and 2) *where* (e.g., at which REST calls) to inject faults. The former determines the effectiveness and validity of the injected faults, while the latter also affects test efficiency. Rainmaker's fault injection policies are guided by a bug taxonomy we developed to describe how error handling could go wrong under the cloud-based fault model. The taxonomy is simple: it considers transient error(s) that can occur during *one* REST API call initiated by the application (and the corresponding retries by the SDK); yet, it captures common bug patterns and shows that error (mis)handling of even one REST call can have major impacts on application correctness. Rainmaker uses a small set of injections to cover all taxonomized bug patterns.

Rainmaker also enables efficient testing to achieve high testing coverage with a small number of test runs. Rainmaker employs automatic dynamic instrumentation to record the application's calling context of each REST API call and to inject call-site information in the HTTP header of outgoing requests; Rainmaker's HTTP-layer fault injection uses the information to selectively inject faults based on a desired code coverage metric. The calling context also enables Rainmaker to build diagnosis support to help developers debug application behavior under fault injection (when a bug is detected).

Finally, Rainmaker includes automatic oracles to flag a fault-injection test outcome as a likely bug, with low false positives. The oracles utilize exceptions and assertions of existing software tests. For exceptions, Rainmaker does not naïvely report any exception that fails a test as a bug, but checks whether the exception is consistent with the injected fault—an inconsistent exception indicates that the fault was handled intentionally, but inappropriately (at least insufficiently).

Key results. We have implemented Rainmaker for .NET applications. Rainmaker supports a number of cloud services: Azure Storage (including Blob Storage [7], Queue Storage [11], and Table Storage [12]), Azure CosmosDB [9], AWS Simple Storage (S3) [1], and AWS Simple Queue (SQS) [4]. Supporting a new cloud service only takes the configuration of the SDK API namespace and the request-ID tag. Rainmaker is fully transparent to the application under test. We evaluate Rainmaker with 11 popular .NET applications that use the supported cloud services. Rainmaker found 73 new bugs in total, among which 55 have been confirmed, and 51 have been fixed (after we reported them). Many of the bugs have severe consequences, such as unexpected application termination, data loss/inaccessibility, and resource leaks. Rainmaker's test oracles are mostly accurate, with a very low false-positive rate (1.96%), making its test results trustworthy.

Summary. The paper makes the following contributions:

- We unravel emerging reliability challenges of cloud-based programming, faced by cloud-backed applications, under the existing design of cloud service APIs and SDKs;
- We present a taxonomy to systematically understand error-handling bugs that render cloud-backed applications vulnerable to transient errors under the cloud-based fault model;
- We develop Rainmaker, the first push-button reliability testing technique for cloud-backed applications, which can effectively and efficiently detect bugs of myriad patterns;
- We have made Rainmaker publicly available at `https://github.com/xlab-uiuc/rainmaker`, with instructions to reproduce all discovered bugs.

2 Background and Motivation

We discuss the emerging reliability challenges faced by cloud-backed applications as the background and motivation of our work. Ideally, cloud-based programming should *not* be different from traditional application programming using native libraries. Unfortunately, as we will show in this section, this is rarely the case in practice—handling errors under the cloud-based fault model is challenging and error-prone.

2.1 Errors in Cloud-backed Applications

There are three key components related to how cloud service related errors are exposed to the applications.

Error responses from cloud services. A request from the application can fail due to a client-side error (e.g., local network timeout) or a service-side error (e.g., temporary service

SDK	Retry (HTTP Status Codes)	(API)	Notes
Azure Storage (Blob/Queue/Table)	408, 429, 500, 502, 503, 504	Any	Only 429 and 503 are retried before v12.3.0
Azure CosmosDB (HTTP mode)	403, 404, 408, 503	Read	Only enabled for multi-region (no retry for single-region)
AWS S3 / AWS SQS	500, 502, 503, 504	Any	Inconsistencies in different lang. SDKs (e.g., Java versus .NET)

Table 1: The retry policies of different cloud services. Besides the HTTP status codes, SDKs could also retry on error messages, e.g., the Azure Storage SDK also retries on messages including `InternalError`, `OperationTimedOut`, and `ServerBusy`.

unavailability). In the latter case, the service returns an error response indicating the error type. Most cloud services are RESTful, and their APIs reuse HTTP response status codes defined by the HTTP/1.1 standard. HTTP status codes 4XX and 5XX indicate "client errors" and "server errors" respectively. In addition, a cloud service can include service-specific error codes in response payload to indicate fine-grained error types. For example, Azure Blob Storage can return 44 different error codes (e.g., `BlobImmutableDueToPolicy` and `BlobAlreadyExists`) with the same HTTP status code 409 (`Conflict`) [8]. The practice is also used by other cloud services, e.g., CosmosDB [15], S3 [3], and SQS [2]. Applications use these codes to understand the nature of the errors and take error-handling actions accordingly.

Retry on transient errors. When a request fails due to a *transient* client- or service-side error (e.g., network timeout and server overload), an application may retry the request, hoping that it would eventually succeed. Cloud-backed applications mostly use the SDKs provided by the cloud service providers to interact with the cloud services. Besides offering easy-to-use, expressive APIs, SDKs also include error handling logic with the goal of providing a native programming experience. For example, when a REST API call to a cloud service fails due to a transient error, the SDK tries to mask the error from the application by retrying the request [67].

Propagating errors up to the application. If the retry efforts fail or if the error is of a *permanent* type, the SDK propagates the error up to the application in a way that is consistent with a native programming experience. For example, .NET and Java SDKs propagate errors to applications as exceptions.

2.2 Emerging Reliability Challenges

2.2.1 A lack of standards and consistencies

Our analysis of multiple cloud service APIs and SDKs from Azure and AWS reveals a lack of standards and behavior consistencies in all three components above, across cloud services and SDKs from the same/different cloud providers.

Unanticipated errors. We observe many undocumented and inconsistent error codes returned by cloud services. For example, as of September 2022, common HTTP error codes such as 408 (`RequestTimeout`) and 429 (`TooManyRequests`) that can be returned by Azure Table Storage [78] are absent from its official documentation [66]. We also observe that the same error can be represented by different error codes across services. For example, Azure Queue Storage represents the error `QueueNotFound` by the code 404, while AWS SQS uses the code 400 for the same error. We even reported and fixed multiple typos in error messages in Azure Storage SDKs.

Second, whether an error will be masked by the SDK (with retries) and whether an error will be propagated to an application vary widely across different services, different versions of the same service, and different language support of the same version. Table 1 shows that SDKs of different cloud services implement different retry policies. Azure Storage .NET SDKs before v12.3.0 only retry on error codes 429 and 503; the later versions add retry for 408, 500, 502, and 504 [28]. The .NET SDK for AWS retries on `LimitExceededException`, but the Java SDK does not. Finally, whether an error is propagated to the application varies even across SDKs from the same provider. The `DeleteMessage` API of Azure Queue propagates an exception to the application when a 404 is returned by the service. On the other hand, the `DeleteEntityAsync` API of Azure Table silently ignores 404 errors and returns success.

The inconsistencies make it hard for developers to anticipate whether a cloud service will return a specific error and whether the SDK will propagate it to the application.

Note that, unlike libraries and system services for traditional applications, a cloud service neither is a part of the application nor has standard APIs such as POSIX.

Retry and non-idempotent APIs. While retry is a common practice to mask transient errors, the retry may introduce subtle errors. In particular, a retry on a non-idempotent cloud service API can cause elusive effects, such as remote data corruption, which can remain latent or lead to additional errors (more details in §3). However, we observe that each service implements different retry logic, with no standard practice or discipline. As a result, the semantic of SDK APIs, under error conditions, is often opaque and inconsistent.

2.2.2 Rarity and large space of faults

Developers often miss error-handling bugs with small-scale, short-duration functional tests because cloud-based faults, which could expose such bugs, are rare. Fault-injection tools allow developers to simulate error scenarios during testing. However, although it is not hard to implement fault-injection mechanisms [35, 44], the key challenges lie in specifying *policies* about what faults to inject, where and when to inject them, and *oracles* about what post-fault conditions indicate a likely bug for developers to inspect. The space of possible fault policies and oracles is large, if not infinite.

2.3 Our Goal

Our goal is to address the emerging reliability challenges faced by cloud-backed applications by (1) systematically understanding the bug patterns, and (2) building practical tooling to systematically test whether a cloud-backed application can correctly handle the myriad errors that may happen during interactions with the cloud services it depends on. We emphasize a "push-button" technique that can be directly used by application developers in an existing testing environment, without the need of writing additional code or configurations.

3 Bug Taxonomy

To systematically understand the patterns of error handling bugs of cloud-backed applications and to guide the design of the Rainmaker tool, we develop a bug taxonomy to describe how error handling could go wrong under the cloud-based fault model. Figure 2 depicts the bug taxonomy as a tree. An important trait is that the taxonomy considers transient error(s) that occurred during *one* REST API call interaction initiated by the application. It does not reason about multiple independent REST API calls.

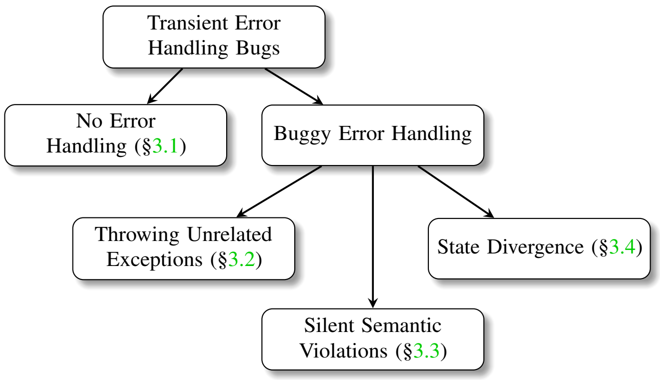
3.1 No Error Handling

In this pattern, the application simply does not handle certain transient errors. This can happen due to the inconsistencies mentioned in §2.2.1. An application may not anticipate a cloud service to return a specific error or may mistakenly expect the SDK to mask a known transient error. For example, an application using Azure Storage's .NET SDKs before v12.3.0, which do not retry on certain transient error codes, may mistakenly assume that *all* transient errors are masked by the SDK and hence not handle them. As a result, some transient error from the cloud service can lead to application crashes or other undesirable behavior.

3.2 Throwing Unrelated Exceptions

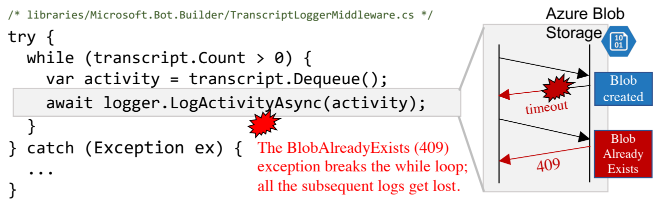
In this pattern, (mis)handling of an error results in a new unhandled error that is usually *unrelated* to the root-cause fault. A common example of this pattern involves request retries. When a request to a cloud service fails with a timeout, the SDK or the application cannot determine whether the timeout happened on the request path or on the response path. SDKs commonly treat timeout as a transient failure and retry. However, if the timeout happened on the response path (in which case, the original request was executed by the service successfully), the retry can fail with a new error (different from the original error) since the original request has invalidated the precondition of the retry. This new error, if not handled properly, can lead to undesirable effects.

Figure 3 shows an example of such bugs [32] detected by Rainmaker in Microsoft BotBuilder [17]. BotBuilder stores logs in the Azure Blob Storage service. Each log operation



Transient Error Handling Bugs
No Error Handling (§3.1)
Buggy Error Handling
Throwing Unrelated Exceptions (§3.2)
State Divergence (§3.4)
Silent Semantic Violations (§3.3)

Figure 2: Taxonomy of error handling bugs in cloud-backed applications. The taxonomy addresses the handling logic of transient error(s) that occurred during the interaction of one REST API call.



```
/* libraries/Microsoft.Bot.Builder/TranscriptLoggerMiddleware.cs */
try {
  while (transcript.Count > 0) {
    var activity = transcript.Dequeue();
    await logger.LogActivityAsync(activity);
  }
} catch (Exception ex) {
  ...
}
```

The BlobAlreadyExists (409) exception breaks the while loop; all the subsequent logs get lost.
Azure Blob Storage
Blob created
timeout
Blob Already Exists
409

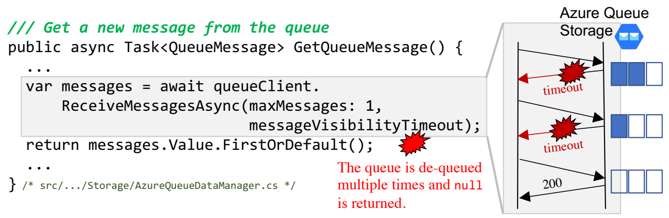
Figure 3: A bug of throwing unrelated exceptions in Microsoft BotBuilder detected by Rainmaker (confirmed and fixed). The Azure Storage SDK automatically retries on timeouts, which returns a 409 error because its precondition is invalidated by the first request.

calls an SDK API to create a new blob. If the API call successfully creates the blob, but the response times out, the Azure Storage SDK automatically retries the request (§2). However, since the blob has already been created by the first request, the retry operation fails with a 409 (`BlobAlreadyExists`) error. The SDK propagates this *permanent* error to the application. BotBuilder does not anticipate or handle the error. This breaks the execution of a loop that is supposed to upload a list of logs to the Blob Storage service, resulting in a loss of subsequent log data. Note that the exception seen by the application is unrelated to the root cause (a transient timeout).

In the next two categories, the buggy error handling does not immediately throw an exception. Rather, it causes unexpected (local or remote) state changes that may cause visible symptoms (e.g., exceptions) during subsequent execution.

3.3 Silent Semantic Violations

In this pattern, mishandling of a transient error causes semantic violations of the REST API specification, without observable symptoms. The REST call returns successfully, and hence the application executes in a happy path. However, the silent semantic violation may eventually result in data loss/corruption, or other incorrect application behavior. One common example of this pattern is manifested by a similar root cause as the one in §3.2: response timeout. Differently,



```
/// Get a new message from the queue
public async Task<QueueMessage> GetQueueMessage() {
  ...
  var messages = await queueClient.
      ReceiveMessagesAsync(maxMessages: 1,
                           messageVisibilityTimeout);
  return messages.Value.FirstOrDefault();
  ...
} /* src/.../Storage/AzureQueueDataManager.cs */
```

Figure 4: A silent semantic violation bug in Microsoft Orleans detected by Rainmaker. Azure Storage SDK automatically retries multiple times on timeouts, and mistakenly empties the queue.

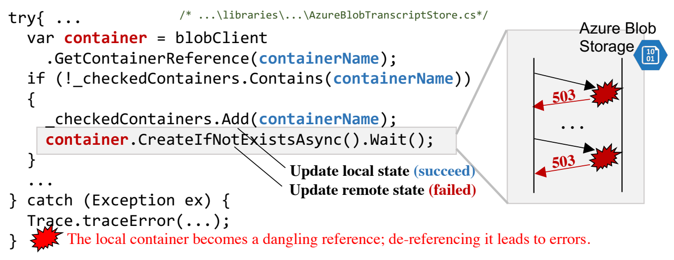
in this pattern, the retry operation succeeds, and the SDK successfully hides the transient error from the application.

Figure 4 shows a silent semantic violation [71] detected by Rainmaker in Microsoft Orleans [13]. Orleans uses Azure Queue Storage service to manage messages. It implements a method `GetQueueMessage()` to dequeue one message from the queue. As shown in Figure 4, the method calls the SDK API `ReceiveMessagesAsync` to dequeue a message from the remote service. The corresponding HTTP request is non-idempotent and should not be naïvely retried [76], because each retry changes the contents of the queue. However, the SDK API automatically retries the request on a transient fault. If a request successfully dequeues a message but its response times out, the SDK retries the request multiple times, each of which, if successful, can dequeue a message. If the last retry/response succeeds, the API successfully returns the message. The application does not handle this corner case, even though the API documentation mentions it. Such behavior violates the semantic of the `GetQueueMessage()` API which is documented to only "*get a message from the queue*" [62]. In fact, repeated retries can dequeue all the messages from the queue, in which case the SDK API returns `null`; Orleans does not expect such behavior and would dereference the `null` pointer and crash. Note that `ReceiveMessagesAsync` is not the only method that has such behavior. If we replace it with `SendMessageAsync`, the above example can enqueue more messages than expected, which may lead to silent resource leaks on the cloud service. Such silent semantic violations are hard for application developers to fix or even detect, as the retries are done by the SDK and are agnostic to the application.

3.4 State Divergence

In this pattern, a mishandled transient error leads to divergence of the local state (in the application) and the remote state (in the cloud service). There is no API semantic violation as in §3.3, but the state divergence could lead to undesired application behavior, e.g., exceptions and resource leaks.

State divergence can happen when an application *optimistically* updates a local state that is correlated with the success of a cloud API call made after the update. The bug manifests if a transient fault fails the request (so no change on the cloud side), but the application does not restore the optimistic up-



```
try{ ...          /* ...\libraries\...\AzureBlobTranscriptStore.cs*/
  var container = blobClient
    .GetContainerReference(containerName);
  if (!_checkedContainers.Contains(containerName))
  {
    _checkedContainers.Add(containerName);
    container.CreateIfNotExistsAsync().Wait();
  }
  ...
} catch (Exception ex) {
  Trace.traceError(...);
}
```

Azure Blob Storage
503
...
503
Update local state (succeed)
Update remote state (failed)
The local container becomes a dangling reference; de-referencing it leads to errors.

Figure 5: A local-state divergence bug in Microsoft BotBuilder detected by Rainmaker (confirmed and fixed). Azure Storage SDK automatically retries multiple times (for 503).

date. The updated state makes the application behave as if the REST API call succeeded, while it actually failed.

Figure 5 shows a state-divergence bug [30] from Microsoft BotBuilder [17] detected by Rainmaker. BotBuilder uses Azure Blob Storage to store blob data which is organized into containers. To create a container, BotBuilder calls REST API `CreateBlobContainer`. When transient errors (e.g., 503 `ServerBusy` errors) occur on the request path of a `CreateBlobContainer` call, BotBuilder swallows the exception in the `catch` block. However, BotBuilder adds the container into its local state of created containers *before* calling `CreateBlobContainer`. As a result, the local state is corrupted with a dangling container pointer, which leads to crashes when BotBuilder dereferences the pointer (e.g., with a list operation). The bug has the same essence as file system bugs that violate update dependencies [39]. On the other hand, transient errors are likely more frequent than file system crashes.

State divergence can also happen when a request changes the remote state, but the application is unaware of the change. Such bugs can manifest when a transient fault breaks the *return path* of a REST call that has changed the remote state. If not handled correctly, the application would assume the call never succeeded, leading to inconsistencies of states.

4 Rainmaker

4.1 Overview

Rainmaker is a "push-button" reliability testing tool for applications that use RESTful cloud services, such as Azure Storage, Azure CosmosDB, and AWS S3. It checks whether the application under test can correctly tolerate or handle common transient errors under the cloud-based fault model (e.g., temporary service unavailability and request timeouts), and detects bugs like the ones described in §3. Its "push-button" nature comes from the automatic fault-injection policies (§4.2) and oracles (§4.3), which are generic and applicable to any application that uses the supported cloud services.

A developer can directly apply Rainmaker as a "plugin-and-play" tool to their existing test suites in their existing testing environments, without writing additional code or configurations. The plugin-and-play nature is achieved by its fault injection mechanism—Rainmaker injects errors by intercepting outbound REST API calls made by the application to the cloud service at the HTTP layer. It includes a standalone HTTP proxy component to do the interception. For example, to inject a 5XX error on the request path of a REST call, the proxy blocks the request from reaching the service and responds with an HTTP response containing the 5XX error code. To inject a timeout on the response path, the proxy lets the request go to the service; however, on receiving the response, it introduces a delay to force a timeout at the application.

Compared with injecting faults directly into application code (e.g., in the forms of exceptions), intercepting at the HTTP layer brings a number of technical benefits: 1) it allows intercepting *fine-grained* REST calls made by the application, including those triggered by SDK retries. As we discussed in §3, injecting errors into retry requests and responses is crucial for exposing certain categories of bugs; 2) error handling in cloud-backed applications depends on not only exception types but also HTTP status codes; 3) it makes the fault injection mechanism transparent to the application under test and work irrespective of the application language and architecture; 4) HTTP requests/responses are highly interpretable, and errors can be uniformly injected by manipulating HTTP responses, with no need to understand external dependencies (e.g., complex exception objects with multiple fields).

Usage. Rainmaker takes as input an existing testing suite that uses RESTful cloud services such as Azure Storage services or their emulators [68], and a desired coverage metric (§4.2.2). Rainmaker first installs a local HTTP proxy that can intercept and manipulate HTTP traffic to and from cloud services (or their emulators). It then selects and executes a minimal set of tests required to achieve the target coverage. As tests are executed, Rainmaker injects faults into their REST API calls according to automatic fault-injection policies. After each test is executed, Rainmaker's oracles analyze the test outcomes and raise alerts as potential bugs are detected.

We envision Rainmaker to be a standard, widely-used testing utility as a part of cloud service SDKs.

4.2 Fault Injection Policy

A fault injection policy specifies *what* faults to inject and *where* (at which REST API calls) to inject them. Rainmaker's fault injection policies are designed with two main objectives.

First, the policies should be *effective*. This requires them to 1) inject only *valid* faults and 2) cover the myriad bug patterns of the taxonomy (§3). One common policy is *randomized* fault injection (e.g., selecting a random fault at a random REST call). However, randomized injection can hardly be effective. As shown in §3, to expose certain bug patterns needs multiple specific faults injected along the interaction of a REST API call—randomized injection is unlikely to hit the specifics. In fact, randomized injection cannot even guarantee valid faults. For example, returning a 503 (`ServiceUnavailable`) error after a write request is successfully executed is invalid, because this is inconsistent with the cloud service contract.[1]

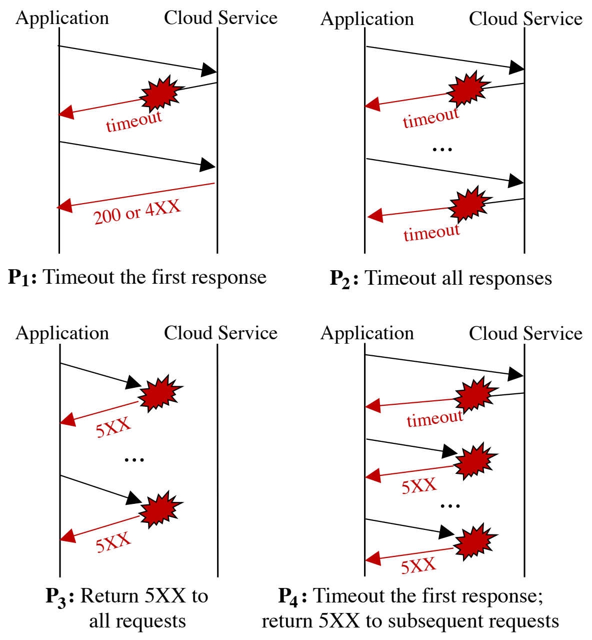


Figure 6: Fault injection policies of Rainmaker that cover all the bug patterns in our taxonomy (see Table 2). Arrows represent HTTP requests and responses for a *single* REST call (and retries). 5XX represents an error code for transient service-side failure. For a REST call with no retry, the four policies are reduced to two.

Second, the policies should enable *efficient* testing—achieving high testing coverage with a small number of test runs. Exhaustively injecting all possible faults at every REST call could be prohibitively expensive, because one test could issue thousands of REST API calls (see §5.3), and many different faults are possible for each call. This is further aggravated by the fact that each fault injection may require a separate test run because injecting the first fault might disrupt a test's subsequent execution.

We next discuss how Rainmaker achieves the two objectives in §4.2.1 and §4.2.2, respectively. Note that Rainmaker can be easily extended to support new policies.

4.2.1 What faults to inject (for a REST API call)?

Rainmaker injects transient faults that occur during the interaction of one REST API call, following the taxonomy in §3. However, the fault space is large even for a single REST call. This is because each of the large number of possible faults may occur on the request or the response path of the original request or subsequent retires issued by the SDK. Interestingly, we find that a small set of four policies (Figure 6) are sufficient to cover the taxonomized bug patterns, as shown in Table 2. For REST calls that do not retry, the four policies are

[1]In practice, a cloud service backend can have bugs to return such an inconsistent response [23]. However, we do not consider buggy cloud services.

Bug Pattern	Fault Injection Policies
No error handling	P_1, P_2, P_3, P_4
Throwing unrelated exception	P_1
Silent semantic violation	P_1, P_2
State divergence	P_1, P_2, P_3, P_4

Table 2: The mapping from the bug patterns and the error injection policies that can potentially expose each bug pattern.

reduced to two: 1) return a transient error code to the request, and 2) timeout the response. The four policies are:

- **P_1** (Timeout the first response). This policy forces a retry that can expose bugs related to invalidated preconditions. Since the timeout is at the response path after the request takes effect at the cloud service, the retry could trigger bugs of throwing unrelated exceptions, silent semantic violation, and/or state divergence, depending on the REST API semantics and the handling logic.
- **P_2** (Timeout all responses). This policy presents an entirely timed-out REST call to the application, while the REST call has taken effects (potentially multiple times) at the cloud service. It could trigger bugs of silent semantic violation and state divergence.
- **P_3** (Return transient error codes to all requests). Under this policy, the REST call does not reach the cloud service as all requests are returned with transient service-side errors. The policy can potentially expose bugs of state divergence, if the local state is optimistically changed before the REST call, but not restored after the call fails.
- **P_4** (Timeout the first response and return transient error codes to all subsequent requests). This policy presents a failed REST call (with a transient error code in response) to the application, which on the contrary has been successfully executed exactly once at the cloud service. It could trigger bugs of state divergence.

Rainmaker by default injects 503 (`ServiceUnavailable`), as the transient fault(s) for request failures. For the responses, Rainmaker injects a timeout fault. These two types of transient faults are common and safe to inject and thus are valid faults on the request and response path, respectively. In comparison, error codes like 500 (`InternalServerError`) have undefined semantics and service-side behavior. The error code can be further customized for specific cloud services and their SDKs based on their definitions of transient faults and retry policies.

Rainmaker identifies all the retried requests and their responses of each REST API call by checking the unique request ID in the HTTP header (e.g., `x-ms-client-request-id` for Azure Storage services). The request ID is provided by the SDK to identify the specific REST call request and is shared by all the subsequent retried requests and responses.

One design choice we make is to avoid encoding specifications of REST/SDK APIs in policies (e.g., idempotency of a REST API and retry behavior of an SDK API). Leveraging API information can help optimize test efficiency. However, it is known that specifications are expensive to maintain and are often incomplete and outdated in practice. Rainmaker minimizes its assumption on the REST/SDK APIs.

	Coverage Metric
C_1	Cover all tests; for each test, select the first REST call.
C_2	Cover all unique call sites of the application code; if a call site is exercised by multiple tests, select the cheapest test.
C_3	Cover all tests and all unique call sites in a pairwise manner.
C_4	Cover all unique call sites of every test; if multiple REST calls exercises the same call site, select the first call to inject.

Table 3: The coverage metrics supported by Rainmaker. We use C_4 as the default metric. Note that injecting faults in every REST call in every test is prohibitive (see §5.3).

4.2.2 Which REST API calls to inject faults?

A test suite may generate an excessive number of REST calls; injecting faults into all of them can be prohibitively expensive, even with the above optimized policies (it could take several machine-months for one application, §5.3). In fact, many REST calls can be redundant (e.g., invoked by the same application code location) for the purpose of covering new error-handling code. Rainmaker therefore selectively injects faults into a small number of REST calls, which achieves certain coverage metrics and optimizes testing resources.

Coverage metrics. Rainmaker supports four different coverage metrics (Table 3). While C_1 measures coverage in terms of the REST calls, C_2–C_4 involves call sites of cloud service APIs. We use the term *call site* to denote a location in the application code that invokes an SDK API that eventually makes one or more REST calls (typically, one SDK API invokes one REST API [26, 27]). Call sites reside in the application code, while REST calls are constructed by the SDK. Hence, C_2–C_4 are more intuitive to developers than C_1.

However, computing C_2–C_4 is challenging for Rainmaker and for any other tools that inject faults via a separate HTTP proxy [37, 47]. This is because the proxy process does not have visibility of call sites within the test/application process.

Making call sites available to the HTTP proxy. Rainmaker addresses this challenge with two techniques using automated instrumentation. First, Rainmaker enables a test to communicate with the HTTP proxy through headers of outgoing HTTP messages. Given test binaries, Rainmaker automatically instruments their outbound HTTP calls. An instrumented call can put its call site information in the header of an outbound HTTP request, so the HTTP proxy can retrieve the information. This can be done automatically since outbound HTTP calls are usually made with a small number of standard core APIs. For example, one needs to instrument only four HTTP client API families (e.g., `HttpClient.SendAsync`) provided by .NET core libraries to intercept outgoing HTTP calls from *all* Azure SDKs (see §4.5).

Second, an outbound HTTP call needs to automatically find its call site to put in the HTTP header. If the application were single-threaded, the instrumented HTTP call could identify the call site by taking the caller of the bottom-most SDK function in the call stack. But, modern applications are multi-threaded, and an HTTP call usually happens asynchronously in a child thread created as a result of executing the call site. In this case, a call stack taken at an outbound HTTP call does not capture the call site that resides in a different thread.

To solve this problem, Rainmaker utilizes inheritable thread-local storage (ITLS) supported by modern languages such as .NET [64] and Java [49]. Any data stored in the current thread's ITLS automatically propagates to all its child threads. Rainmaker automatically instruments all call sites (identified by SDK namespace) so that at runtime, they store their location information in the current thread's ITLS. When a call site eventually invokes an instrumented, outgoing HTTP call, in the same thread or in a child thread, the call retrieves the call site information from its ITLS and puts it in the HTTP header for the proxy process to examine.

Test planning. With the call site information available at the HTTP proxy, Rainmaker can inject faults according to the specified coverage metric in Table 3. Each coverage metric is a tradeoff between completeness and cost.

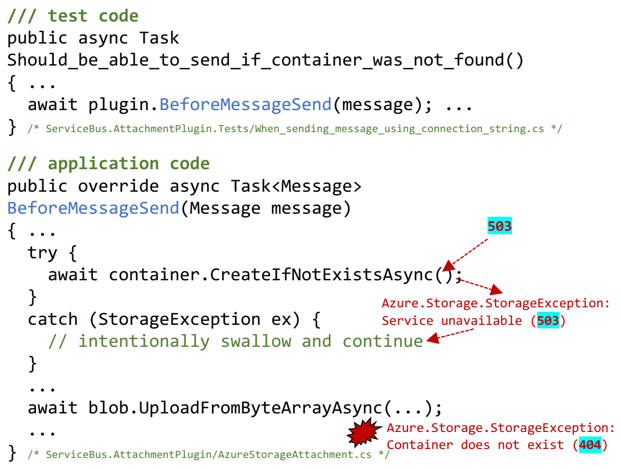
To plan the fault injection runs, Rainmaker first performs a *reference run* of the test suite with no fault injection. During the reference run, Rainmaker measures the time taken by each test and observes, by using its HTTP proxy, the REST calls made by different tests. It then selects the tests that issue REST calls as candidate tests for fault injection. Rainmaker then performs an offline analysis to generate test plans containing the minimum number of fault-injection target REST calls (and their tests) in order to achieve target coverages. It also outputs an approximate running time of each plan using the time of the tests in the reference run and, if any, the delays to be injected to create timeout errors. The time helps a developer choose the right coverage metric by understanding the tradeoff between completeness and cost.

Given the time taken by each test and the set of REST calls each test makes in the reference run, it is straightforward to implement the policies C_1, C_2, and C_4. For C_3, Rainmaker models it as a linear programming (LP) problem of generating a set of pairs that cover all N tests and M unique call sites (with each test covering a subset of call sites), while minimizing the total test running time (each test has different running time). It then uses an LP solver to generate a plan.

Note that test planning, including the reference run and LP solving, is done offline as a one-time effort. The results can potentially be reused across test runs in CI/CD environments.

4.3 Test Oracles

A test oracle checks whether the outcome of a fault injection test run indicates a bug. A trustworthy oracle catches



```
/// test code
public async Task
Should_be_able_to_send_if_container_was_not_found()
{ ...
  await plugin.BeforeMessageSend(message); ...
} /* ServiceBus.AttachmentPlugin.Tests/When_sending_message_using_connection_string.cs */

/// application code
public override async Task<Message>
BeforeMessageSend(Message message)
{ ...
  try {
    await container.CreateIfNotExistsAsync();
  }
  catch (StorageException ex) {
    // intentionally swallow and continue
  }
  ...
  await blob.UploadFromByteArrayAsync(...);
  ...
} /* ServiceBus.AttachmentPlugin/AzureStorageAttachment.cs */
```

503
Azure.Storage.StorageException:
Service unavailable (503)
Azure.Storage.StorageException:
Container does not exist (404)

Figure 7: An exception captured by Rainmaker to detect a state-divergence bug in ServiceBus AttachmentPlugin. The exception that fails the test (404) is inconsistent with the injected fault (503).

different types of bugs with no false alarms so that developers can focus their investigation only on true bugs.

With the goal of being generic and widely applicable to any cloud-backed application, Rainmaker does not use any application-specific oracle. Instead, it devises a set of application-agnostic oracles on top of the existing test oracles encoded in developers' test code. These oracles are effective in identifying various types of bugs, with low false positives.

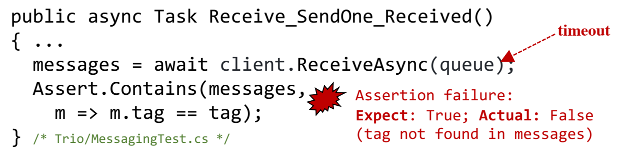
4.3.1 Exception Oracle

This oracle flags a fault-injection test outcome as a potential bug if 1) the test fails with an exception, 2) the exception is created in application code rather than in test code, and 3) the exception is inconsistent with the injected fault. We now explain the rationale behind the three conditions.

When a test fails with an exception as a result of an injected fault, it may not always mean a bug. For example, in the applications we use for our evaluation, many tests directly interact with a cloud service (e.g., to setup the test environment) but without proper error handling. If Rainmaker injects faults into such REST calls, the test will fail with an exception. However, the failure does not indicate application bugs. Rainmaker applies the second condition to filter out test failures due to exceptions created in test code, based on the exception call stack. Note that Rainmaker avoids injecting faults into REST calls with call sites from the test code.

However, not all test failures due to exceptions created in application code are bugs. For example, a utility method of an application may intentionally propagate an exception to the upper layer and expect it to be handled there. When a test for this utility fails due to not handling the exception, the test failure is expected, and it does not indicate a bug.

Rainmaker applies the third condition to only report bugs when the final exception that causes the test failure is inconsistent with the injected fault (by searching the injected HTTP



```
public async Task Receive_SendOne_Received()
{ ...
  messages = await client.ReceiveAsync(queue);
  Assert.Contains(messages,
    m => m.tag == tag);
} /* Trio/MessagingTest.cs */
```

```
timeout
Assertion failure:
Expect: True; Actual: False
(tag not found in messages)
```

Figure 8: An assertion utilized by Rainmaker to detect a bug of silent semantic violation in Storage.NET backed by AWS SQS. When timeouts are injected, `ReceiveAsync` dequeues multiple times, causing an empty message list returned, which fails the assertion.

status code or error code in the exception stack). The intuition is that, if the test fails due to an error different from the injected fault, it indicates that the fault was once handled (it shows the developer's intention to handle it), but the handling is inappropriate (at least insufficient) and causes a different error that fails the test. This oracle can capture bugs of throwing unrelated exceptions as well as silent bugs of semantic violations and state divergence which do not cause immediate exceptions but result in exceptions eventually. Figure 7 shows such an example, where a 503 fault injected to `CreateIfNotExistsAsync` leads to a 404 exception from `UploadFromByteArrayAsync` and fails the test.

Note that the oracle is incomplete. If an application misses error handling (§3.1), exceptions exposed by the test could be a bug. However, at the unit test level, it is indeterminate.

Relaxing the oracle. With all three conditions, the oracle above is conservative. One can relax it to identify other types of likely bugs. If the developer is testing an application or running a system test (instead of a unit test), she can disable the third condition so that Rainmaker flags any failure (e.g., crash) due to exceptions from appliation code as a bug. This is because Rainmaker only injects transient faults that are expected to be handled gracefully.

4.3.2 Assertion Violation Oracle

This oracle flags a fault-injection test outcome as a potential bug if the test fails due to an assertion violation (*and* not an unhandled exception). Intuitively, transient faults injected by Rainmaker should not impair semantic correctness of application code; hence existing assertions should not be violated if the faults are properly handled. The assertions in test code could be brittle to fault injection [45], i.e., an assertion violation is not a bug, but caused by the fault injection changing application runtime behavior. In practice, we find such brittle assertions are small in numbers, as discussed in §5.2. The assertion oracle can capture bugs of silent semantic violations and state divergence. Figure 8 shows how Rainmaker leverages the existing assertion to capture a silent semantic violation in Storage.NET [19].

4.4 Diagnosis Support

Associating source-code information with faults. Diagnosing and localizing bugs in application code triggered by HTTP-level faults can be challenging without source-code context. This can be true even when the developer knows the fault-injected REST API or SDK API (via instrumentation), because they can be invoked by multiple program locations.

To help developers debug the test failures, Rainmaker associates the fault injection with source-code information in the form of the call site, together with the call stack of runtime exceptions or assertions (§4.3). We find the REST API call site and exception/assertion call stack are critical to debugging. They help developers to understand what and where fault(s) were injected and reason about the error propagation inside application code. One can further apply existing techniques to automatically reconstruct the failure execution (e.g., [85]).

Reproducing bugs. Rainmaker can reproduce a reported bug because all fault injections for a test are determined by the test planner (§4.2.2). If an injected fault exposes a bug, Rainmaker can rerun the planned fault injection to reproduce the triggered bug. If the test is nondeterministic, it may take several runs to reproduce the bug. In our experience, the error handling behavior for REST calls is typically local to the call and is rarely affected by nondeterminism of test execution.

4.5 Implementation

We have implemented Rainmaker for Windows. Its HTTP interception is implemented using MockServer [18], with fault-injection policies implemented as MockServer rules in Java. Rainmaker registers a system proxy for Windows Internet Services to forward all HTTP traffic to the MockServer proxy. This enables Rainmaker to inject faults to *any* application that issues REST APIs. The oracles are implemented in Python, which analyzes the raw test results and logs.

Rainmaker currently supports coverage metrics C_2–C_4 for .NET applications only, which needs dynamic instrumentation to record and propagate call site information (§4.2.2). The instrumentation is implemented using the .NET profiling API [65] that enables changing bytecode of a method before it is JITed. Rainmaker inserts call site information in HTTP headers by instrumenting four HTTP API families from the .NET core library that cover all outgoing HTTP messages. We believe the same mechanism can be implemented for Java.

To support a new cloud service in Rainmaker only takes two inputs in the form of configurations: 1) the SDK namespace (e.g., `Azure.Storage*` for Azure Storage SDK) and 2) the request-ID tag of the cloud service (e.g., `x-ms-client-request-id` for Azure Storage services and `amz-sdk-invocation-id` for AWS S3). The former instructs Rainmaker what call sites to record, and the latter identifies a request and its retries (they all have the same request ID).

5 Evaluation

Our evaluation addresses the following questions: 1) Can Rainmaker find new bugs in real-world cloud-backed applications? 2) Are Rainmaker's testing results trustworthy? 3)

Application	Cloud Service	# Stars	# LOC	Selected Tests
Alpakka	Queue	106	14K	9
AttachmentPlugin	Blob	67	1.3K	32
BotBuilder	Blob, Queue	758	18K	67
DistributedLock	Blob	838	17.1K	30
EF Core	CosmosDB	11.7K	842.4K	420
FHIR Server	CosmosDB	897	112.8K	202
Insights	Blob, Queue, Table	20	51.7K	147
IronPigeon	Blob	255	5.4K	7
Orleans	Blob, Queue, Table	8.8K	187K	155
Sleet	S3	276	18.7K	2
Storage.NET	Blob, Queue, S3, SQS	567	12.4K	36

Table 4: The cloud-backed applications used in our evaluation.

What is the tradeoff between running time and coverage?

- §5.1: Rainmaker finds 73 new bugs in all 11 evaluated cloud-backed applications, which represent a swathe of reliability issues. So far, 55 of them have been confirmed, and 51 have been fixed by the developers.
- §5.2: Rainmaker's test oracles have a low false-positive rate (1.96%) with regards to test failures.
- §5.3: Rainmaker significantly reduces running time compared to exhaustive fault injection with coverage guarantee.

Evaluation setup. We evaluated Rainmaker on 11 popular .NET applications that use six different cloud services from two cloud service providers, Azure and AWS (Table 4). These applications are mature and widely used; many are maintained by software companies, such as Orleans, BotBuilder, EF Core, FHIR Server from Microsoft, Insights from NuGet, and Alpakka from Petabridge. They use six cloud services: Blob Storage [7], Queue Storage [11], Table Storage [12], and CosmosDB [9] from Azure, and Simple Storage Service (S3) [1] and Simple Queue Service (SQS) [4] from AWS. We configure Rainmaker to support these services (§4.5).

We apply Rainmaker to existing test suites of the applications. Rainmaker automatically selects tests that interact with the cloud services from the test suite by monitoring HTTP traffic during the reference run (§4.2.2). The number of selected tests varies from 2 to 420 across the applications (Table 4), including both unit and system tests. We differentiate unit and system tests based on their naming conventions.

All the tests that interact with Azure cloud services are run with emulators: Azurite [68] for Blob, Queue, and Table and the CosmosDB emulator [63] for CosmosDB. The tests that interact with AWS are run with the real S3/SQS services; we did not find an official AWS emulator.

5.1 Finding New Bugs

Rainmaker finds a total of 73 *new* bugs in the evaluated applications (Table 5). Those bugs include all the bug patterns in the taxonomy (§3): 29 bugs of no error handling, 23 bugs of throwing unrelated exceptions, four bugs of silent semantic violations, and 17 bugs of state divergence. Rainmaker finds bugs in *every* application, showing the error-proneness of handling transient faults with cloud-based programming. We have reported 66 (out of 73) bugs. So far, 55 of them have been confirmed, and 51 of them have been fixed.

Application	No Error Handling	Throw New Exception	Semantic Violation	State Divergence	Total
Alpakka	0	0	1	1	2
AttachmentPlugin	0	0	0	2	2
BotBuilder	0	2	0	2	4
DistributedLock	0	2	0	0	2
EF Core	7	0	0	0	7
FHIR Server	11	0	0	0	11
Insights	0	10	0	0	10
IronPigeon	0	1	0	0	1
Orleans	0	5	2	11	18
Sleet	0	2	0	0	2
Storage.NET	11	1	1	1	14
Total	29	23	4	17	73

Table 5: New bugs detected by Rainmaker across the applications.

Many of the detected bugs have severe consequences, such as unexpected application termination, data loss/inaccessibility, and resource leaks (Table 6). All these consequences are triggered by transient faults against *one* REST API call.

Rainmaker detects bugs that are unlikely to be exposed by randomized fault injection. For example, a bug [36] in DistributedLock is only triggered when timeout happens to the response of a specific SDK API call site. The test used for detecting this bug issues 900+ requests in total; only four requests are from that specific call site. Rainmaker can consistently detect this bug as it systematically exercises the call sites of the REST API calls (§4.2.2).

Table 7(a) shows that all four fault injection policies (Figure 6) employed by Rainmaker are effective in finding bugs. The four policies address different fault scenarios and are complementary to each other. No policy detects all the bugs. Similarly, Table 7(b) shows that oracles are complementary to each other. For example, all the semantic violation bugs are captured by assertions as they do not cause exceptions.

The 73 bugs cause 2,654 test failures in total. To inspect them, we cluster test failures based on (a) the application call site that invokes the REST API where fault(s) are injected and (b) the exception stack trace in the application namespace, or assertion. For two test failures with both (a) and (b) being the same, they are considered to have the same root cause, i.e., injecting to the same API call site causes the same exception stack trace in application namespace or assertion violation.

No error handling. Rainmaker found 29 bugs of this type, where neither the SDK nor the application handles the injected faults. These bugs are all from applications that use Azure Storage SDK with versions before 12.3.0 and the CosmosDB SDK; the former does not retry timeouts, and the latter does not retry with our single-region setting. The applications using these SDKs are expected to handle transient errors but do not have error handling. These bugs are manifested in system tests when Rainmaker injects faults into the REST calls.

Consequence	Example	# Bugs
Externalizing unrelated exception	IronPigeon-133: Deletion operations perform on non-existent resources [46].	36
Operation failure	AttachmentPlugin-277: Containers cannot be created due to transient error [24].	35
Incorrect results or states	BotBuilder-5787: The timestamp metadata of blobs is not set correctly [31].	13
Application crash	FHIR Server-2732: FHIR Server can crash unexpectedly upon transient faults [38].	11
Data loss or inaccessibility	BotBuilder-6407: Activities that should be logged are lost [32].	4
Resource leak	Orleans-7790: Redundant messages were added incorrectly to the Queue service [72].	2

Table 6: Consequences of the bugs found by Rainmaker. One bug can lead to multiple consequences.

Application	Fault Injection Policy				Test Oracle		
	P_1	P_2	P_3	P_4	Exp (Unit)	Exp (Sys)	Assert.
Alpakka	2	1	1	0	0	0	2
AttachmentPlugin	0	1	2	1	2	0	0
BotBuilder	2	1	2	1	3	0	1
DistributedLock	2	0	0	0	2	0	0
EF Core	7	n/a	7	n/a	0	7	0
FHIR Server	11	n/a	11	n/a	0	11	0
Insights	10	0	0	0	10	0	0
IronPigeon	1	0	0	0	1	0	0
Orleans	17	11	11	11	16	0	2
Sleet	2	0	0	0	2	0	0
Storage.NET	13	12	12	12	2	11	1
Total	67	26	46	25	38	29	6

Table 7: The breakdown of the number of bugs captured by the fault injection policies (left) and oracles (right). For EF Core and FHIR Server, the four policies are reduced to two, because CosmosDB SDK does not retry in our setting (single region). For the exception oracle ("Exp"), we differentiate unit tests and system tests. Note that one bug can be exposed by multiple fault injection policies.

Throwing unrelated exceptions. Rainmaker found 23 bugs of this type. Rainmaker triggers these bugs by injecting timeout to the response path after a request takes effect at the cloud service (see Figure 3). While the bug in Figure 3 is captured by an assertion on the number of created blobs, many other bugs are captured by the exception oracle (the exception is mostly inconsistent with the injected fault). Such bugs can be avoided if the cloud service can collapse the retries: If the first attempt is successful, the cloud service should ignore the following retries of the same SDK API call. This requires the cloud service to identify the retries of each SDK API call, which can be specified by the SDK when it issues a retry.

Silent semantic violations. Rainmaker found four bugs of this type. All of them are caused by retrying non-idempotent REST APIs (e.g., `ReceiveMessagesAsync` in Figure 4). Rainmaker detects these bugs by injecting timeout to trigger non-idempotent retries and leveraging assertions to catch semantic violations (as in Figure 8). Different from traditional system services (e.g., file systems), cloud services seldom have standard API specifications like POSIX for file system APIs; documents are often outdated or incomplete. Without precisely understanding the semantic and side effect of each REST API, it is difficult for developers to avoid silent semantic violations.

State divergence. Rainmaker found 17 bugs of this type. Some applications maintain local data structures to reflect the state of the remote resources hosted by the cloud service. Rainmaker triggers state divergence by injecting 5XX error codes to the request path or timeout to the response path (e.g., Figure 5). Although the inconsistencies do not immediately lead to exceptions, Rainmaker can still catch them when the test throws exceptions (e.g., Figure 7) or fails assertions.

5.2 False Positives

While identifying bugs with its test oracles, Rainmaker introduces a very low false positive rate of 1.96% (52/2,654). It reports in total 2,654 test failures for the evaluated applications. Among the failures reported, only 52 of them were false alarms. The low false positive rate is attributed to Rainmaker's exception oracles (see §4.3.1). If Rainmaker directly reports exceptions thrown by unit tests, it would have reported 3.07 times more test failures. To validate that the oracles filter out little true alarms, we randomly sample a hundred exceptions that are filtered out by Rainmaker's exception oracle and find that none of them indicates a bug.

Among the 52 false alarms, 10 of them come from five tests of Insights. These tests exercise scenarios where a client issues invalid requests and expects the return of certain error codes (e.g., 404 `Not Found`). Since Rainmaker injects 5XX on the request path (P_3 and P_4 in Figure 6), the REST call fails by assertions on the original error codes. To avoid those false alarms needs to understand those REST calls, e.g., based on information from the reference run. Note that Rainmaker does not inject faults on an HTTP response that already has an error code.

The other 42 false alarms were caused by 14 tests from Alpakka and IronPigeon. Those tests use a small connection timeout that was exceeded due to the fault injection, resulting in either assertion failures or inconsistent exceptions. These tests are considered flaky tests in software testing literature [54]. Strictly speaking, flaky tests should not be counted as false alarms. However, detecting flaky tests is not a goal of Rainmaker, and the flakiness is indeed triggered by fault injection (it can also be triggered by slow connections).

5.3 Running Time with Coverage

Rainmaker takes 0.57–212.77 hours to test each application under coverage metric, C_4 (Table 3), as shown in Table 8. All the experiments were run on a Windows 10 Pro with AMD Ryzen 9 5900X 3.70 GHz CPU and 32 GB memory. Over 93.54% of the running time is spent on executing fault injection test runs. Rainmaker also spends 0.02–16.61 hours to 1) conduct the vanilla run and 2) generate the test plan. The test plan generation using a linear programming (LP) solver takes only 1.46–3.67 seconds across the applications and is negligible compared with the time for vanilla test runs. The numbers of constraints and variables range from 14 to 241 and

Application	Running Time (Machine Hours) Test Planning	Fault Injection	Total	# Fault Injection Test Runs
Alpakka	0.02	0.97	0.99	196
AttachmentPlugin	0.04	2.83	2.87	416
BotBuilder	0.32	10.36	10.68	888
DistributedLock	0.13	4.97	5.10	572
EF Core	10.65	159.60	170.25	4786
FHIR Server	16.61	196.16	212.77	6428
Insights	0.36	10.97	11.33	3056
IronPigeon	0.06	0.51	0.57	120
Orleans	2.57	53.17	55.74	3804
Sleet	0.11	0.63	0.74	72
Storage.NET	2.30	40.27	42.57	1512

Table 8: Running time of Rainmaker in machine hours, for the most expensive coverage metric C_4. Test planning includes both the vanilla test run and test plan generation (the latter takes seconds).

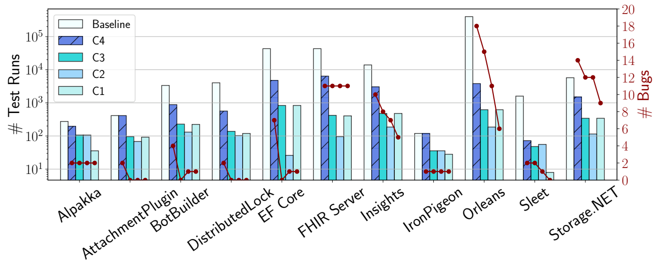


Figure 9: The number of fault injection test runs of each coverage metric, C_1–C_4 in Table 3 (bars) and the number of bugs detected by each coverage metric (dots). Baseline refers to exhaustively injecting faults to all the REST API calls of all the tests.

from 18 to 3214, respectively. Note that test plan generation is a one-time effort and is done offline; fault injection takes multiple test runs and is more time-consuming.

Rainmaker's test planning avoids exhaustively injecting faults in every REST call (the baseline). Figure 9 shows the number of fault injection test runs each coverage metric can reduce, compared to the baseline. Overall, Rainmaker with C_4 (default coverage) reduces on average 64.47% of test runs for each application compared to the baseline. In particular, C_4 reduces 394,172 (99.04%) test runs for Orleans alone because Orleans has stress tests that repeatedly exercise the same SDK API call site in large loops. C_4 reduces fault injection runs by "deduplicating" REST calls with the same call site. Without the reduction, Rainmaker would take 588 days to test Orleans.

In terms of bug finding effectiveness, C_4 covers all the tests and REST calls that are covered by C_1–C_3. Thus, C_4 detects all the bugs that are detected by C_1–C_3.

6 Discussion and Limitation

Rainmaker's effectiveness depends on the adequacy of the existing test suite, in particular, tests that interact with cloud services. Such tests may not be abundant (Table 4). For example, in Microsoft Orleans, only 155 out of 7,002 tests interact with cloud services. A more common testing practice is to mock REST APIs [53]. Our future work is to auto-rewrite mocked tests into tests that can be utilized by Rainmaker.

Rainmaker is not cheap. It may need to run a test multiple times, each injecting different fault(s) or to a different REST call. For big test suites, Rainmaker would need significant machine hours (Table 8). In fact, our evaluation shows that many bugs trigger multiple test failures (§5.1). A future work is to reduce the cost using test selection techniques [83].

Rainmaker currently only targets faults that occur in one interaction of REST API call initiated by the application and the subsequent retries (§3). We are investigating how to inject faults to multiple *correlated* API call interactions which has a larger fault space and a more complex fault model.

Rainmaker can be extended to test applications under potential cloud service bugs. In our evaluation, we find that cloud services have various correctness issues. For example, we observe that the same REST API of AWS S3 has different consistency guarantees at different regions [25], which can break application assumptions. Also, the error behavior is often opaque and hard to reason, e.g., the side effect of a non-idempotent API call when it returns 500 (`InternalServerError`).

Not every bug found by Rainmaker is easy to fix. For example, timeouts on the response path make it hard for SDK/application to know whether the failed request has taken effect at the cloud service. Server-side support such as versioning (e.g., based on HTTP ETag) and transaction-like API support could potentially help non-idempotent APIs. Moreover, SDKs should not blindly retry non-idempotent APIs, which however is not an uncommon practice, as shown in Table 1.

Our current prototype of Rainmaker focuses on .NET applications. We believe that the prototype can be generally extended to support applications in other languages as well. The high-level idea of injecting faults with an HTTP proxy is independent of the programming language of the target application. The only .NET-specific component in our prototype is the one that computes the coverage metrics C_2, C_3, and C_4 (in Table 3) by using .NET profiling API [65] for dynamic instrumentation and .NET inheritable thread-local storage (ITLS) [64] for propagating call-site information. Neither dynamic instrumentation nor ITLS is unique to .NET; they are already available or can be supported in other languages and runtimes such as Java.

7 Related Work

Our work focuses on push-button tooling to help developers address emerging reliability challenges of cloud-backed applications. The techniques that embody Rainmaker have lineages of error-handling analysis and fault injection.

- **Error-handling code analysis.** It is known that error handling is a main root cause of production failures of software systems [40, 41, 84]. Prior work developed static analysis for error-handling code to search missing logs and `TODO`s in error-handling code [84, 86], check error specifications [48], and understand error propagation [41, 77, 82].

- **Fault injection.** Prior work developed fault injection techniques for traditional software applications [29, 35, 59, 87] and for distributed backend systems (e.g., storage and data processing systems) that empower modern cloud services [16, 20, 21, 33, 34, 40, 43, 51, 55–57, 69, 73, 80]. They implicitly or explicitly target error-handling code. Moreover, many existing fault injection tools support injection at the HTTP layer [37, 47].

Unfortunately, we realized that none of the above techniques support a push-button tool that is generic, widely applicable to cloud-backed applications because they all have one or more of the following limitations. First, many fault injection tools only provide mechanisms without fully automatic policies beyond randomization or oracles beyond crashes [6, 21, 35, 40, 43, 44, 51, 58, 61, 73, 75]. Developers need to manually implement them, which is nontrivial. Second, many techniques use application or domain knowledge to devise policies and oracles [16,20,33,34,55,56,69,80]. For example, fault injection for distributed databases checks consistency and isolation properties by injecting node crashes and network partitions [16, 52]. These techniques cannot be used as a common, basic utility for diverse types of applications. Third, fault injection at the program level [35, 50, 59, 60, 87] is fundamentally limited to cloud-backed applications: 1) program errors (exceptions and errno) are too coarse-grained to expose certain bug patterns (e.g., silent semantic violations) which need fine-grained injection at the HTTP layer; 2) it is nontrivial to construct exception objects—error handling of cloud-backed applications is based on not only exception types, but also HTTP status codes (Table 1); 3) few program fault injection considers cloud states, but assumes a single program. Rainmaker instead injects faults at the REST interface, effectively addressing the above three limitations.

The early form of cloud-backed applications is mobile apps that interact with cloud backends via REST APIs. Unlike today's cloud services, the cloud backend is specific to an app and is not widely used as a building block of generic application development. Most fault injection tools for mobile apps focus on GUI testing [74, 79]; few considers app-cloud interactions. Rainmaker applies to cloud-backed mobile apps.

8 Concluding Remarks

Rainmaker serves as a first step tooling to help developers test application reliability under the cloud-based fault model conveniently, when writing cloud-backed code. Despite being a simple tool, Rainmaker can effectively detect bugs in many existing cloud-backed applications, indicating the challenge and error-proneness of correctly handling transient errors. Our goal is to make Rainmaker a basic utility running against every cloud-backed application to detect critical bugs at development time. We hope to inspire more advanced, specialized tooling and raise discussions on cloud service support and SDK designs to eliminate reliability threats in the first place.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Raja Sambasivan, for their insightful comments. We thank Shan Lu, Darko Marinov, Lalith Suresh, and Jun Zeng for valuable feedback and discussions that helped improve this work. We thank Yifei Song for his help on the evaluation and Shuai Wang and Jinghao Jia for helping with the machine setup. We thank all the cloud-backed application developers who engaged with us and reviewed our reports and patches. This work was funded in part by NSF CNS-2130560, SHF-1816615, CNS-2145295, and Microsoft Azure credits.

References

[1] Amazon S3. https://aws.amazon.com/s3/, 2022.

[2] Amazon Simple Queue Service Common Errors. https://docs.aws.amazon.com/AWSSimpleQueueService/latest/APIReference/CommonErrors.html, 2022.

[3] Amazon Simple Storage Service Error Responses. https://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html, 2022.

[4] Amazon SQS. https://aws.amazon.com/sqs/, 2022.

[5] AWS Cloud Products. https://aws.amazon.com/products, 2022.

[6] AWS Fault Injection Simulator. https://aws.amazon.com/fis/, 2022.

[7] Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/, 2022.

[8] Azure Blob Storage error codes. https://learn.microsoft.com/en-us/rest/api/storageservices/blob-service-error-codes, 2022.

[9] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/, 2022.

[10] Azure products. https://azure.microsoft.com/en-us/products, 2022.

[11] Azure Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/, 2022.

[12] Azure Table Storage. https://azure.microsoft.com/en-us/services/storage/tables/, 2022.

[13] dotnet/orleans. https://github.com/dotnet/orleans, 2022.

[14] Google Cloud products. https://cloud.google.com/products, 2022.

[15] HTTP Status Codes for Azure Cosmos DB. https://docs.microsoft.com/en-us/rest/api/cosmos-db/http-status-codes-for-cosmosdb, 2022.

[16] Jepsen. https://jepsen.io/, 2022.

[17] microsoft/botbuilder-dotnet. https://github.com/microsoft/botbuilder-dotnet, 2022.

[18] MockServer. https://www.mock-server.com/, 2022.

[19] Storage.NET. https://github.com/aloneguid/storage, 2022.

[20] ALAGAPPAN, R., GANESAN, A., PATEL, Y., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[21] ALQURAAN, A., TAKRURI, H., ALFATAFTA, M., AND AL-KISWANY, S. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[22] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., LOO, B. T., AND OUTHRED, G. 007: Democratically Finding the Cause of Packet Drops. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Apr. 2018).

[23] ATLIDAKIS, V., GODEFROID, P., AND POLISHCHUK, M. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'19)* (May 2019).

[24] ATTACHMENTPLUGIN-277. Transient error happening in container existence check will lead to container not found exception. https://github.com/SeanFeldman/ServiceBus.AttachmentPlugin/issues/277, 2022.

[25] AWS-SDK-NET-2084. The retry request of PutBucket will behave differently between regions. https://github.com/aws/aws-sdk-net/discussions/2084, 2022.

[26] AWS-SDK-NET-2102. 1-to-N Mappings between SDK API and REST APIs. https://github.com/aws/aws-sdk-net/discussions/2102, 2022.

[27] AZURE/AZURE-SDK-FOR-NET-31001. 1-to-N Mappings between SDK API and REST APIs. https://github.com/Azure/azure-sdk-for-net/issues/31001, 2022.

[28] AZURE/AZURE-SDK-FOR-NET-9670. Retry on 408, 500, 502, 504 status codes. https://github.com/Azure/azure-sdk-for-net/pull/9670, 2022.

[29] BANABIC, R., AND CANDEA, G. Fast Black-Box Testing of System Recovery Code. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys'12)* (Apr. 2012).

[30] BOTBUILDER-DOTNET-5778. _checkedContainers can become inconsistent with blob storage and further lead to unhandled exception. https://github.com/microsoft/botbuilder-dotnet/issues/5778, 2021.

[31] BOTBUILDER-DOTNET-5787. Metadata of blob can be missing due to transient error which leads to unhandled exception. https://github.com/microsoft/botbuilder-dotnet/issues/5787, 2022.

[32] BOTBUILDER-DOTNET-6407. Activities that should be logged are missing due to transient network errors. https://github.com/microsoft/botbuilder-dotnet/issues/6407, 2022.

[33] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)* (Apr. 2012).

[34] CHEN, H., DOU, W., WANG, D., AND QIN, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)* (Sept. 2020).

[35] CHRISTAKIS, M., EMMISBERGER, P., GODEFROID, P., AND MÜLLER, P. A General Framework for Dynamic Stub Injection. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)* (May 2017).

[36] DISTRIBUTEDLOCK-132. Transient error leads to unhandled 409 when releasing an Azure lease. https://github.com/madelson/DistributedLock/issues/132, 2022.

[37] ENVOY DOCS. Envoy Fault Injection. https://www.envoyproxy.io/docs/envoy/latest/api-v3/extensions/filters/http/fault/v3/fault.proto, 2022.

[38] FHIR SERVER-2732. Retry other HTTP error codes from Cosmos DB? https://github.com/microsoft/fhir-server/issues/2732, 2022.

[39] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS) 18*, 2 (May 2000), 127–153.

[40] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. Fate and Destini: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[41] GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIBLIT, B. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)* (Feb. 2008).

[42] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)* (Aug. 2015).

[43] HEORHIADI, V., RAJAGOPALAN, S., JAMJOOM, H., REITER, M. K., AND SEKAR, V. Gremlin: Systematic Resilience Testing of Microservices. In *Proceedings of the IEEE 36th International Conference on Distributed Computing Systems (ICDCS'16)* (June 2016).

[44] HUNT, G., AND BRUBACHER, D. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium* (July 1999).

[45] HUO, C., AND CLAUSE, J. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)* (Nov. 2014).

[46] IRONPIGEON-133. make blob deletion tolerant of transient errors. https://github.com/AArnott/IronPigeon/pull/133, 2022.

[47] ISTIO DOCS. Istio Fault Injection. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/, 2022.

[48] JANA, S., KANG, Y., OHIO, S. R., AND RAY, B. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016).

[49] JAVA API SPECIFICATION. Class InheritableThreadLocal<T>. https://docs.oracle.com/javase/7/docs/api/java/lang/InheritableThreadLocal.html, 2022.

[50] JIANG, Z.-M., BAI, J.-J., LU, K., AND HU, S.-M. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *Proceedings of the 29th USENIX Security Symposium* (Aug. 2020).

[51] JU, X., SOARES, L., SHIN, K. G., RYU, K. D., AND SILVA, D. D. On Fault Resilience of OpenStack. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SOCC'13)* (Oct. 2013).

[52] KINGSBURY, K., AND ALVARO, P. Elle: Inferring Isolation Anomalies from Experimental Observations. In *Proceedings of the VLDB Endowment* (Nov. 2020).

[53] KRYMETS, P. Unit testing and mocking with Azure SDK .NET. https://devblogs.microsoft.com/azure-sdk/unit-testing-and-mocking/, 2020.

[54] LAM, W., MUŞLU, K., SAJNANI, H., AND THUMMALAPENTA, S. A Study on the Lifecycle of Flaky Tests. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)* (May 2020).

[55] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[56] LU, J., LIU, C., LI, L., FENG, X., TAN, F., YANG, J., AND YOU, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (Oct. 2019).

[57] MAJUMDAR, R., AND NIKSIC, F. Why is Random Testing Effective for Partition Tolerance Bugs? In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[58] MARINESCU, P. D., BANABIC, R., AND CANDEA, G. An Extensible Technique for High-Precision Testing of Recovery Code. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)* (June 2010).

[59] MARINESCU, P. D., AND CANDEA, G. LFI: A Practical and General Library-Level Fault Injector. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)* (June 2009).

[60] MARINESCU, P. D., AND CANDEA, G. Efficient Testing of Recovery Code Using Fault Injection. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (Dec. 2011), 1–38.

[61] MEIKLEJOHN, C. S., ESTRADA, A., SONG, Y., MILLER, H., AND PADHYE, R. Service-Level Fault Injection Testing. In *Proceedings of the 2013 ACM Symposium on Cloud Computing (SOCC'21)* (Nov. 2021).

[62] MICROSOFT DOCS. AzureQueueDataManager.GetQueueMessage Method. https://learn.microsoft.com/en-us/dotnet/api/orleans.azureutils.azurequeuedatamanager.getqueuemessage, 2022.

[63] MICROSOFT DOCS. Install and use the Azure Cosmos DB Emulator for local development and testing. https://learn.microsoft.com/en-us/azure/cosmos-db/local-emulator, 2022.

[64] MICROSOFT DOCS. .NET CallContext Class. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.remoting.messaging.callcontext, 2022.

[65] MICROSOFT DOCS. .NET profiling. https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview, 2022.

[66] MICROSOFT DOCS. Table Storage error codes. https://learn.microsoft.com/en-us/rest/api/storageservices/table-service-error-codes, 2022.

[67] MICROSOFT DOCS. Transient fault handling. https://learn.microsoft.com/en-us/azure/architecture/best-practices/transient-faults, 2022.

[68] MICROSOFT DOCS. Use the Azurite emulator for local Azure Storage development. https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azurite, 2022.

[69] MOHAN, J., MARTINEZ, A., PONNAPALLI, S., RAJU, P., AND CHIDAMBARAM, V. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[70] NUGET. WindowsAzure.Storage NuGet. https://www.nuget.org/packages/WindowsAzure.Storage, 2022.

[71] ORLEANS-7738. Popping up queue messages may cause data loss and unexpected NullReferenceException. https://github.com/dotnet/orleans/issues/7738, 2022.

[72] ORLEANS-7790. Data was added repeatedly to the queue unexpectedly without any warning. https://github.com/dotnet/orleans/issues/7790, 2022.

[73] PILLAI, T. S., CHIDAMBARAM, V., KISWANY, S. A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[74] RAVINDRANATH, L., NATH, S., PADHYE, J., AND BALAKRISHNAN, H. Automatic and Scalable Fault Detection for Mobile Applications. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)* (June 2014).

[75] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)* (May 2006).

[76] RFC 9110 HTTP SEMANTICS. Idempotent Methods. https://www.rfc-editor.org/rfc/rfc9110.html#name-idempotent-methods, 2022.

[77] RUBIO-GONZÁLEZ, C., GUNAWI, H. S., LIBLIT, B., ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. Error Propagation Analysis for File Systems. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI'09)* (June 2009).

[78] STACKOVERFLOW-39661635. Retry on 408 Timeout from Azure Table Storage service. https://stackoverflow.com/questions//retry-on-408-timeout-from-azure-table-storage-service, 2016.

[79] SUN, J., SU, T., LI, J., DONG, Z., PU, G., XIE, T., AND SU, Z. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the 2021 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)* (July 2021).

[80] SUN, X., LUO, W., GU, J. T., GANESAN, A., ALAGAPPAN, R., GASCH, M., SURESH, L., AND XU, T. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[81] TAN, C., JIN, Z., GUO, C., ZHANG, T., WU, H., DENG, K., BI, D., AND XIANG, D. NetBouncer: Active Device and Link Failure Localization in Data Center Networks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)* (Feb. 2019).

[82] WEIMER, W., AND NECULA, G. C. Finding and Preventing Run-Time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)* (Oct. 2004).

[83] YOO, S., AND HARMAN, M. Regression Testing Minimisation, Selection and Prioritization: A Survey. *Software Testing, Verification, and Reliability 22*, 2 (Mar. 2012), 67–120.

[84] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G. R., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[85] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the 15th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-XV)* (Mar. 2010).

[86] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., TANG, X., ZHOU, Y., AND SAVAGE, S. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Oct. 2012).

[87] ZHANG, P., AND ELBAUM, S. Amplifying Tests to Validate Exception Handling Code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)* (June 2012).

[88] ZHANG, Q., YU, G., GUO, C., DANG, Y., SWANSON, N., YANG, X., YAO, R., , CHINTALAPATI, M., KRISHNAMURTHY, A., AND ANDERSON, T. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Apr. 2018).