UNDERSTANDING AND IMPROVING THE FIDELITY OF LOCAL DEVELOPER TESTING FOR CLOUD-BASED SOFTWARE

BY

ANNA MAZHAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Assistant Professor Tianyin Xu

ABSTRACT

Modern software projects have been progressing towards a cloud-based, serverless programming model, where software applications use cloud services as important components. Such cloud-based programming practice greatly simplifies software development by harvesting cloud benefits (e.g., high availability and elasticity). However, it imposes significant challenges for software testing and analysis, due to the opaqueness of cloud backends and the monetary cost of invoking cloud services for continuous integration and deployment. As a result, software developers commonly use cloud emulators for offline development and testing, before online testing and deployment.

This thesis presents a systematic analysis of cloud emulators from the perspective of cloud-based software testing. Our goal is to (1) understand the discrepancies introduced by cloud emulation with regard to software quality assurance and deployment safety and (2) address inevitable gaps between emulated and real cloud services. The analysis results are concerning. Among 255 APIs of five cloud services from Azure and Amazon Web Services (AWS), we detected discrepant behavior between the emulated and real services in 94 (37%) of the APIs. These discrepancies lead to inconsistent testing results, threatening deployment safety, introducing false alarms, and creating debuggability issues. The root causes are diverse, including accidental implementation defects and essential emulation challenges. We discuss potential solutions and develop a practical mitigation technique to address discrepancies of cloud emulators for software testing.

”Dedicated with love to my parents and siblings, whose support is immeasurable; to my husband, who stands by me; and to my friends, who walk this journey with me”

ACKNOWLEDGMENTS

I would like to begin by expressing my gratitude to those who have supported me throughout the process of completing this thesis. Firstly, I would like to extend my deepest thanks to my advisor, Professor Tianyin Xu. When I joined UIUC, I was filled with self-doubt and was hesitant about leading a project, but Tianyin strengthened my confidence and entrusted me with a project in collaboration with Microsoft Research. His belief in my capabilities was a pivotal force in my success. He consistently ensured that he fulfilled his role by providing all necessary opportunities. Tianyin has a very unique relationship with his students; I deeply admire him for building a relationship that made me comfortable to voice all my concerns and feedback, which was invaluable during moments of uncertainty and reservation. His active guidance and high standards in academic research have significantly enhanced my learning, setting a high benchmark for my future endeavors.

I would like to thank all the contributors of the Cloudtest project whose efforts led to a successful conference paper submission. I am grateful to Dr. Suman Nath from Microsoft Research, whose insightful feedback was instrumental in shaping the project. I am immensely grateful to William Zheng and Saad Sher Alam for their dedication, hard work and invaluable contributions.

I had a wonderful time at UIUC, enjoying memorable experiences, engaging in thought-provoking discussions, and forming lasting relationships. This brings me to thank all my colleagues in the lab. I would especially like to extend my appreciation to my xlab peers — Yinfang Chen, Xudong Sun, and Hao Lin — for their invaluable assistance in reviewing my work and offering guidance. Special thanks go to my good friend, Talha Waheed, whose selfless effort spent chasing an impossible deadline is something for which I will be eternally grateful.

I am also thankful to my undergraduate advisors at LUMS, Dr. Ihsan Ayyub Qazi and Dr. Zafar Ayyub Qazi, who nurtured my initial steps into research and laid the foundation for my academic pursuits. Their advice continues to guide me.

Lastly, and most importantly, I owe a profound debt of gratitude to my family. I am grateful to my parents for raising me to be a confident and strong woman. Despite the challenges for a woman in the culture I was born into, they enabled me to pursue every opportunity. I would also like to express my deepest gratitude to my husband, my sanctuary. I met him at UIUC, and it did not take long for me to realize that he is a significant part of my life. Thank you for being my anchor during times of weakness.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

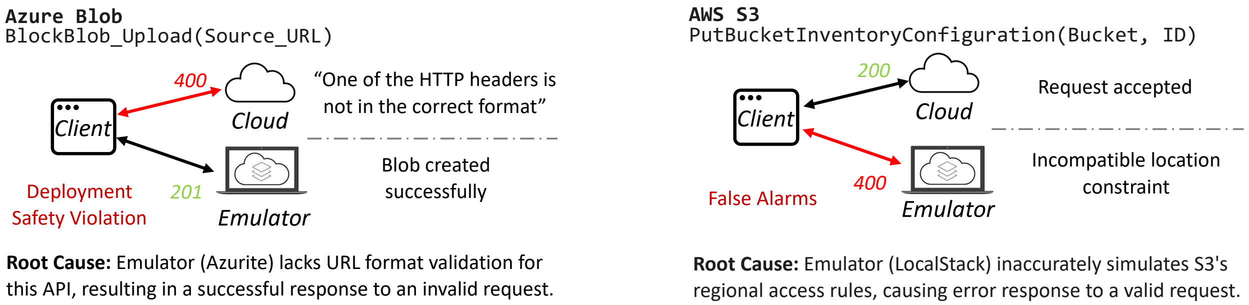
1.1 MOTIVATION

Modern software development has been progressing towards a cloud-based, serverless programming model, where software applications increasingly use cloud services as important components for storage, database, data processing, etc. Such cloud-based programming practice greatly simplifies software development and deployment (e.g., applications no longer needs to purchase or manage large-scale infrastructue) and helps by harvesting cloud benefits (e.g., high availability and elasticity) and software deployment by reducing the cost of purchasing and managing large-scale systems and infrastructures. Today, all major cloud providers offer various services to support cloud-based software [1, 2, 3]. These cloud services are widely used, e.g., the .NET SDK of Azure Storage services alone has tens of thousands of downloads daily [4].

Despite its benefits, cloud-based programming imposes significant challenges to software testing and analysis due to the opaqueness of cloud backends and the monetary cost of invoking cloud services during continuous integration and deployment (CI/CD). First, unlike other types of dependencies like libraries, which are linked as a part of the software program, cloud services are *external* to cloud-based software (invoked via REST API calls), and their backend implementations are opaque. It is hard to reason about the correctness of cloud-based software independently, especially its end-to-end behavior. For example, regressions of cloud backend implementations [5] can directly affect dependent software that invokes corresponding APIs.

Second, testing cloud-based software with cloud services can be costly, especially with CI/CD. Cloud services charge users based on the number of API invocations, storage capacity, and additional features like transaction support [6, 7]. So, extensive testing on the cloud is expensive. For example, the test suite of Orleans (a cloud-based software project) issues 120K+ Azure API calls. Under CI/CD, tests are continuously invoked [8, 9]. We expect even higher costs in the near future as cloud services are increasingly adopted by software projects and new tests are being added.

Today, developers commonly use *cloud emulators* for cloud-based software development and testing before online testing and deployment. Nine out of ten projects we studied (Chapter 3) use emulators for CI tests. A cloud emulator offers local simulation of large, complex cloud services. For example, a fault-tolerant, persistent key-value storage service can be emulated by a centralized, in-memory hash table [10]. Cloud emulators enable developers

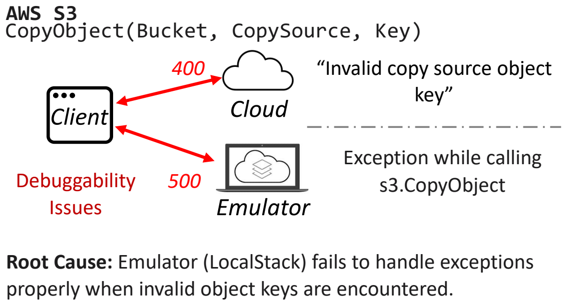


Root Cause: Emulator (Azurite) lacks URL format validation for this API, resulting in a successful response to an invalid request.

Root Cause: Emulator (LocalStack) inaccurately simulates S3's regional access rules, causing error response to a valid request.

(a) Deployment safety violation

(b) False alarm



Root Cause: Emulator (LocalStack) fails to handle exceptions properly when invalid object keys are encountered.

(c) Debuggability issue

Figure 1.1: Implications of discrepancies between cloud emulators and cloud services with regards to software testing.

to conduct prompt, cost-efficient offline testing and debugging [11]. They are transparent to software under test—using emulators requires no code change but a simple setup to connect to emulated services. Cloud emulators are typically developed or supported by cloud service providers. For example, Microsoft provides emulators for Azure services, e.g., Azurite [12] for Azure Storage Services [13].

Ideally, emulators should behave the same as real cloud services so that software quality assurance, like testing, can rely on emulators. However, it is prohibitively difficult for emulators to achieve perfect fidelity (considering the complexity, scale, and distributed nature of cloud services). In practice, emulators implement specifications of cloud service APIs (Chapter 2). However, as shown in our study (Chapter 5.1), specifications of today's cloud services and their APIs are often incomplete and limited. Without formal enforcement of emulator compliance with real cloud service, it is unclear how much fidelity today's emulators could realize. We use the term *discrepancies* to refer to emulator behavior that deviates from expected behavior of cloud services. We observed that discrepancies are constantly reported to affect testing of cloud-based software [14, 15, 16, 17].

In this thesis, we aim to breakdown the fidelity of cloud emulatos and eventually, bridge the gap between cloud emulators and cloud services, aiming to elevate the precision of cloud emulation and its consequential benefits on the quality of cloud-based software and the enhancement of developer experience. Specifically, we meticulously employ differential testing on two renowned cloud emulators—Azurite [12] for Azure Storage services (encompassing Blob, Table, and Queue) and LocalStack [18] for Amazon Web Services (including S3 and DynamoDB)—to spotlight the nuances in behavior between emulated and actual cloud environments. By documenting and scrutinizing these variances, we delve into the underlying root cause behind each discrepancy[1].

Our analysis has yielded results that are very concerning. Among 255 APIs from five cloud services, including Azure and Amazon Web Services (AWS), we detected discrepant behavior between the emulated and real services in 94 (37%) of the APIs. These discrepancies have profound implications for deployment safety and developer experience:

1. Code that passes tests with emulators may fail in production with real cloud services.
2. Test failures with emulators can be false alarms.
3. Debugging with emulators can be challenging due to discrepant feedback (e.g., error codes and messages).

Figure 1.1 shows three examples we discovered in our analysis. We further analyze ten open-source cloud-based software projects; five of them are affected by discrepancies—some of their tests have inconsistent results when running on the cloud emulator versus the cloud services. In one project (Durabletask [19]), 78% of the tests are affected.

The root causes of discrepancies are diverse but can be categorized into (1) incompleteness of existing specifications, (2) unspecified behavior, and (3) implementation defects (such as bugs and missing features). While these root causes reflect essential software engineering challenges, we believe that many discrepancies could be addressed by more comprehensive testing and more systematic specification. We discuss potential solutions and mitigations, ranging from practical formal methods to new system-level support (Chapter 6).

We explore hybrid cloud-emulator testing as a short-term mitigation and develop a simple tool named ET to selectively run tests on emulators versus cloud services, based on whether the test invokes discrepant APIs (Chapter 7). ET offers different policies depending on whether discrepant API information is known as apriori or being determined via in-situ analysis. Through ET, we demonstrate that hybrid testing not only yields considerable cost savings compared to running all tests with cloud services.

---

[1] As a first step, this thesis focuses on basic functional correctness, instead of performance or fault tolerance (e.g., data consistency and crash consistency) which are beyond the expectation of local emulation.

1.2 CONTRIBUTION

This thesis makes the following main contributions:

- An in-depth exploration of the challenges encountered in testing cloud-based applications using cloud emulators. This discussion sheds light on the obstacles that the developers face in ensuring the reliability of cloud software.
- A comprehensive analysis that scrutinizes the discrepancies between actual cloud services and their emulators. This analysis delves into the characteristics of these discrepancies, identifies their underlying causes, and evaluates their impact on software testing.
- A thorough discussion on potential solutions for addressing these discrepancies including a mitigation tool designed to selectively run software tests on the emulator while achieving a balance between reliability and cost-saving in the testing process.
- A detailed account of the bugs that were discovered during the analysis and reported to the developers. So far, six of these bugs have been confirmed and five have been fixed.
- Our research artifact is available for public access at GitHub repository: `https://github.com/team-cloudtest/cloudtest`.

CHAPTER 2: BACKGROUND

2.1 CLOUD SERVICES AND THEIR APIS

Modern cloud services are programmatically accessed via REST APIs on top of HTTP(S). A service typically exposes several tens of REST APIs. REST APIs use HTTP requests along with URIs (Uniform Resource Identifies) for accessing resources on the cloud. It utilises HTTP methods (POST, GET, PUT, DELETE) to perform CRUD operations (Create, Read, Update, Delete) on the resources exposed by the cloud service. For example, AWS S3 exposes 97 REST APIs [20], and Azure Blob service exposes 72 REST APIs [21]. To ease developer programming, cloud services provide Software Development Kits (SDKs) with high-level, language-specific library APIs. Typically, SDK APIs wrap the raw REST APIs, and for a service, more SDK APIs are built on top of the REST APIs. (e.g., S3 Python SDK has 108 APIs [22] and the Python SDK of Azure Blob has 81 APIs [23], respectively). Most existing cloud-based applications invoke SDK APIs to interact with the cloud services instead of calling raw REST APIs. Figure 2.1 shows an example of creating an Azure Blob Container using REST API and its equivalent SDK method in Python.

API Specification. The REST APIs are commonly described using specification languages such as OpenAPI Specification [24]. The specification describes the API version,

```
Request Syntax:
PUT https://myaccount.blob.core.windows.net/mycontainer?restype=container
HTTP/1.1

Request Headers:
x-ms-version: 2011-08-18
x-ms-date: Sun, 25 Sep 2011 22:50:32 GMT
x-ms-meta-Name: StorageSample
Authorization: SharedKey myaccount:{key}
```

Listing 2.1: REST API Request

```
def create_container(self): from azure.storage.blob import BlobServiceClient
    blob_service_client =
    BlobServiceClient.from_connection_string(self.connection_string)

    try: self.blob_service_client.create_container(self.container_name) except
        Exception as e: print(e)
```

Listing 2.2: SDK Method in Python

Figure 2.1: REST API call to create an Azure Blob Container and its equivalent SDK method in Python.

request URI, content type, input parameter, output format, error code and messages, etc. The API specifications are used by cloud emulators (Chapter 2.2) to develop emulated APIs.

We find that the API specifications are often incomplete. For example, the OpenAPI specification of Azure Blob services only specifies value constraints (including data types) for 63 (59%) of 107 parameters across all Azure Blob REST APIs. Moreover, all the specifications are for data type, value range, and default value, without deeper behavior semantics.

SDKs often implement additional checks to constrain parameter values of API calls over the API specifications. Hence, parameter values that can satisfy the API specification can be rejected by the SDK checks to prevent erroneous and malformed REST API calls from the applications.

Pricing. Cloud services are expensive. Despite different pricing models of cloud services, pricing typically depends on the amount of data to be stored and the cost of operations. Take Azure Blob service as an example. The price for 100TB/month ranges from $91–$1,545, depending on the access tiers [6]. Azure Blob service then charges for read, write, iterative-read, and iterative-write operations separately [25]. For example, the price for write operations varies from $0.0228–$0.13 per 10,000 writes, depending on the tier. Other features, such as redundancy [26, 27, 28], further increase the cost.

With the current pricing model, testing cloud-based software incurs non-trivial monetary costs. To demonstrate the cost, we run the tests of Orleans (a cloud-based software project) with the Azure Storage services with standard configuration. Orleans has 189 tests that issue 120K+ Azure API calls over 23 unique APIs. We run these tests 500 times, which costs $74.5 US dollars (we expect 500 times to be a reasonable time in CI/CD of large software projects [29, 30]). Moreover, as cloud and serverless computing trends emerge, applications are expected to increasingly utilize cloud service APIs, leading to significantly higher testing costs.

2.2 CLOUD EMULATOR

To reduce cost and get prompt feedback, developers commonly use cloud emulators for offline development and testing. Developers also commonly use emulators to debug production problems. Emulators run as local daemons that simulate cloud services. Cloud-based software programs transparently interact with the emulator like how they interact with cloud services—using an emulator only needs a simple configuration that switches the connection from a cloud handle to localhost listened to by the emulator; no code change is needed.

Most cloud services provide developers with official emulators. For example, Microsoft provides emulators for Azure Storage and CosmosDB, and AWS provides emulators for DynamoDB and Step Functions. Moreover, third-party emulators are developed. One successful example is LocalStack [18], which emulates many AWS services such as S3 and DynamoDB. Compared with official emulators, LocalStack provides a more usable integrated development environment [31]. Our study deliberately selects an official emulator (Azurite) and a third-party emulator (LocalStack).

For compliance with the target cloud services, cloud emulators are commonly built on top of API specifications. For example, Azurite uses AutoRest [32] to generate stub code from the OpenAPI specification of Azure Storage services [33]. LocalStack employs weekly GitHub Action Checks to detect any changes of the API specifications of AWS [34].

CHAPTER 3: METHODOLOGY

We use differential testing to discover discrepancies between cloud emulators and real cloud services. Basically, we issue the same REST API calls to the emulated service and the cloud service independently and check the resulting behavior, including the return values, error codes or messages (if any), and states of key data objects such as blobs and containers. Any inconsistent behavior is a potential discrepancy. Once we confirm a discrepancy, we debug it and localize the source-code location in the emulator that causes the discrepancies.

3.1 STUDIED EMULATORS AND SERVICES

Emulators. We choose to study Azurite [12] and LocalStack [18] because they represent state-of-the-art emulators and are widely used. Azurite represents the official emulators provided by cloud service providers while LocalStack represents third-party emulators developed by companies of cloud-based integrated programming environments. Importantly, both emulators are open-sourced, which enables us to debug discovered discrepancies. Table 3.1 lists the information of the two emulators. Note that LocalStack supports 70+ AWS services [31]; Table 3.1 only lists services studied in this thesis.

Services. We select five widely used cloud services: Blob, Queue, and Table services from the Azure Storage services and S3 and DynamoDB from AWS. All these services are supported by the corresponding emulators (Table 3.1).

3.2 TEST WORKLOADS

We use two complementary test workloads. First, we leverage API fuzzing to generate sequences of REST API calls. Each API call sequence is a test workload and the workloads cover all the REST APIs provided by the target cloud services. The API fuzzing workloads help us understand the discrepancies in each REST API and characterize a broad range of APIs.

Table 3.1: Emulators and cloud services studied in this thesis

Emulator	**Service**	**LOC**	**#Commits**	**Developer**
Azurite	Blob, Queue, Table	2,591K	1,034	Official
LocalStack	S3, DynamoDB	449K	5,527	Third-party

Note that the fuzzing is done against SDK APIs, not raw REST APIs. In fact, we started from REST API fuzzing using RESTler [35]. However, we found that certain discrepancies are not possible if the software under test uses SDKs which have additional checks (Chapter 2). In practice, developers rarely craft REST API calls directly but mostly call SDK APIs. Since our goal is to understand discrepancies in the context of software development, rather than security analysis [35, 36], we choose to fuzz SDK APIs. Basically, we focus on analyzing discrepancies faced by cloud-based software developers.

We also use the test suites of existing cloud-based software projects as the test workloads. Many tests invoke cloud service APIs. These tests help understand the impact of discrepancies on testing real-world software projects, which is complementary to fuzzing from the API perspective.

3.2.1 Fuzzing SDK APIs.

We implemented a grammar-aware API fuzzer to generate diverse SDK API calls as test workloads. We start from default or predefined parameter values for each SDK API and the fuzzer mutates parameter values based on value constraints defined in OpenAPI specifications of REST APIs (the "grammar"). To do so, we establish the mapping from the parameters of REST APIs to the ones of corresponding SDK APIs; the mapping process is straightforward because SDK APIs are mostly wrappers over raw REST APIs. The grammar-based mutation ensures that generated SDK API calls are mostly valid and can reach emulated or real cloud services. Our fuzzer implements the fuzzing approach of RESTler [35]: (1) inferring producer-consumer dependencies among request types (e.g., "API Y should be called after API X" because Y takes as an input a resource-ID produced by X) and (2) taking dynamic feedback from responses during testing (e.g., learning that "a API Y called after a sequence $X \rightarrow Y$ is refused" and avoiding this combination in the future).

We monitor the response of each API call. If inconsistent responses are returned by the emulator and the cloud services (including both HTTP response status code like 200 and 404, as well as error code and message if the response returns an error), we capture and record the discrepancy and abort the test. Otherwise, we progress to the next API call in the generated sequence. We also check the key data objects before and after the API calls (e.g., the number of blobs for Azure Blob Services) to capture discrepancies with no immediately observable manifestation, such as resource leaks (Chapter 4.2). Those checks are service-specific.

Due to resource constraints (limited cloud education credits), we were only able to conduct our analysis using the Python SDKs of the studied cloud services.

Table 3.2: Cloud-based software projects. "#Tests" refers to tests that invoke cloud services; "#APIs" refers to *unique* APIs.

Project	**Services**	**LOC**	**#Tests**	**#APIs**
Alpakka	Queue	22.5K	9	6
AttachmentPlugin	Blob	1.9K	23	7
DurableTask	Blob, Queue, Table	59.0K	101	30
IdentityAzureTable	Table	85.7K	51	6
Insights	Blob, Queue, Table	144.8K	171	20
IronPigeon	Blob	37.8K	7	8
Orleans	All services but S3	204.8k	247	35
ServiceStack	DynamoDB, S3	756.2K	187	15
Sleet	Blob, S3	21.2K	22	21
Streamstone	Table	4.6K	75	7

3.2.2 Using existing tests.

To understand the impact of discrepancies on real-world software projects, we perform differential testing using test suites of ten open-source projects that use the studied cloud services (Table 3.2). These projects are mature and widely used, developed by companies like Microsoft (Orleans, DurableTask), NuGet (Insights), and PetaBridge (Alpakka), and are actively maintained (using recent versions of APIs). We make sure to select applications that are actively maintained.

In our study, we select tests that interact with the cloud services. The selection is done by monitoring the HTTP traffic of each test in a reference run using the emulator. We check whether a test outputs inconsistent results when running with emulators versus cloud services. Table 3.2 shows the number of tests that invoke cloud services and the number of unique APIs invoked by the test suite of each project.

CHAPTER 4: DISCREPANCY CHARACTERISTICS

4.1 PREVALENCE OF DISCREPANCIES

Our analysis shows that discrepancies are prevalent in the two cloud emulators (Azurite and LocalStack). The five cloud services we studied expose a total of 255 APIs. Among these 255 APIs, our API fuzzer (Chapter 3) discovered discrepancies in 94 (37%). Table 4.1 shows the number of discrepant APIs of each service. Both Azurite and LocalStack have a considerable percentage of discrepant APIs among all the APIs they support and across the emulated services, showing that discrepancies are not specific to one emulator implementation or specific to APIs of a particular service. Rather, emulator fidelity is a common challenge.

These discrepancies have different implications as exemplified in Figure 1.1, including (1) deployment safety violations (1.1a), (2) false alarms (1.1b), and (3) debuggability issues (1.1c). Table 4.2 categorizes the implications of the total 98 discrepancies discovered in the 94 discrepant APIs (one discrepancy can have different implications). Here, deployment safety violations refer to cases where the same API call returns success on the emulator, but returns error on the cloud service; false alarms refer to cases where the same API call returns error from the emulator, but success from the real service. Debuggability issues refer to cases where the same API call returns different responses from the emulator and the cloud service, making it hard to debug the software. The results show diverse implications of these discrepancies. We measure their impacts on real-world test cases in Chapter 4.3.

Surprisingly, we find that 37 of the 94 discrepant APIs are certified by the emulators and considered "fully supported." LocalStack adopts five methods to certify emulated APIs [37, 38], including both internal and external integration tests (e.g., snapshot tests [39]). Despite extensive efforts, 22 (out of 60) discrepant APIs from LocalStack are certified by all five testing methods, while 39 (out of 60) discrepant APIs are certified with at least one test method. On the other hand, Azurite's baseline is to have coverage from atleast 1 unit/integration test for each newly implemented API[40]. They provide a list of implemented and unimplemented features and APIs based on their testing. We found that 15 out of 34 discrepant APIs from Azurite are certified as fully supported by Azurite [41]. The results demonstrate the difficulties that current testing-based methodologies encounter in identifying inconsistencies, highlighting the need for more comprehensive strategies to ensure high fidelity APIs.

Table 4.1: Discrepant APIs with respect to the cloud services

Services	**Emulator**	**Total APIs**	**Discrepant APIs**
Azure Blob	Azurite	72	31 (43%)
Azure Table	Azurite	15	1 (7%)
Azure Queue	Azurite	18	2 (11%)
AWS S3	LocalStack	97	33 (34%)
AWS DynamoDB	LocalStack	53	27 (51%)
Total		255	94 (37%)

Table 4.2: Impacts of discrepancies across emulators.

Impact	**Azurite/Azure**	**LocalStack/AWS**	**Total**
Deployment safety	13	22	35
False alarms	12	33	45
Debuggability issues	9	9	18
Total	34	64	98

Key Finding: Discrepancies between modern cloud emulators and real cloud services are prevalent (discovered in 37% of APIs on average and even in certified APIs). The implications of discrepancies include unsafe deployment, false alarms, and debuggability issues.

Figure 4.1: Key findings of the practical impact of discrepancies.

4.2 DISCREPANCY MANIFESTATIONS

A notable observation is that discrepancies are manifested through not only inconsistent responses to the API calls, but also inconsistent remote, cloud-side states. The latter creates significant challenges to observe and understand discrepancies, especially with short-running test cases. We implemented domain-specific checks to compare the remote states maintained by the emulators and the corresponding cloud services (Chapter 3). For example, we check the states of each container (maintained by the emulators and cloud services) before and after each container-related API call.

Seven discrepancies have the same response to API calls but create inconsistent remote states. For example, when invoking an Azure Blob API, "`Container_Restore` [42]", to recover an early deleted container, both Azurite and the Azure Blob service return the same response; however, the Blob service faithfully restores the deleted container, while Azurite creates a new empty container. Such discrepancies may not be easy to capture without fine-grained checks.

We also find 44 discrepancies that cause inconsistent responses and inconsistent remote states. For example, when calling a Blob API "`BlockBlob_StageBlockFromURL`" [43] with an invalid URL, Azurite succeeds by creating a new blob, while the Blob service fails with `InvalidHeaderValue`.

Key Finding: Discrepancies that only manifested via inconsistent remote states are hard to observe; fine-grained state checks are needed to capture those silent discrepancies.

Figure 4.2: Key findings of the challenges in observing discrepancies.

4.3 IMPACT ON REAL-WORLD TESTS

We measure the impacts of discrepancies on real-world test suites of cloud-based software projects (Table 3.2). The impact is reflected by inconsistent test results when running the same test with the emulator versus the cloud service. We call a test that invokes discrepant APIs a *discrepant test*; a discrepant test may or may not output *discrepant results*. Among the ten projects we evaluated (Table 3.2), we discovered discrepant test results in 50% of them (five projects), as shown in Table 4.3.

Different projects are affected at different levels, ranging from 1% to 100% of tests that invoke cloud services. The variation is attributed to the usage characteristics of the cloud service APIs. Specifically, though we discovered a large number of discrepant APIs (Chapter 4.1), not all these APIs are equally invoked by the test cases. Figure 4.3 depicts the popularity of

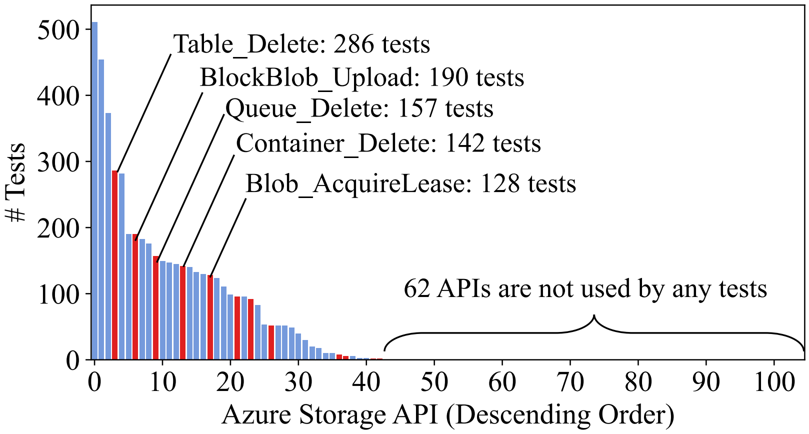


Figure 4.3: Popularity of Azure Storage APIs, measured by the number of tests that use an API across all the studied projects.

Table 4.3: Software tests that are affected by discrepancies.

Project	**# Discrepant Results**	**# Discrepant Tests**	**Impact**	
			Safety Vio.	**False Alarms**
Alpakka	9 (100%)	9	9	0
DurableTask	79 (78%)	101	79	0
Orleans	8 (9%)	82	5	3
ServiceStack	3 (2%)	72	2	1
Streamstone	1 (1%)	75	1	0
Total	100	339	98	4

all Azure Storage APIs (105 in total) invoked by all the tests of the studied projects, where discrepant APIs are marked in red. Popularity is measured by the number of tests that use the API. Among the 105 APIs, only 43 of them are invoked by at least one test. Only 12 APIs (out of 43) involved in the tests are discrepant (while 34 discrepant APIs in total are discovered for Azure; see Table 4.1).

Note that the number of discrepant tests is much larger than the number of discrepant test results manifested (Table 4.1). The reason is that many discrepant tests are only manifested when certain parameter values are used.

The results have two important implications. First, addressing discrepancies can leverage API usage characteristics in the field to prioritize widely used APIs. Oftentimes, fixing discrepancies of a few APIs can eliminate a large number of discrepant tests or test results. We take all the tests using Azure services as the example: by resolving the top five discrepant Azure APIs in Figure 4.3, discrepant tests drop by 63% (from 267 to 99) and discrepant results drop by 10% (from 89 to 80). The small drop in discrepant results is caused by tests in DurableTask utilizing multiple discrepant APIs. If the top seven discrepant Azure APIs are resolved, 75 out of the 79 discrepant results caused by DurableTask will be eliminated.

Second, fine-grained, parameter-level analysis can further capture discrepancies. Although our analysis stays at the API level instead of parameters, we build on these implications when designing mitigation solutions (Chapter 7).

Key Finding: Five out of ten studied software projects reveal discrepant test results caused by discrepancies between emulators and cloud services. Those discrepant tests are caused by a small set of discrepant APIs. Not all discrepant APIs manifest during testing if triggering parameters are not used.

Figure 4.4: Key findings of the discrepancies in real-world test suites.

We further categorize the implications of discrepant tests into (1) *deployment safety violations* (1.1a), and (2) *false alarms* (1.1b), as broken down in Table 4.3. Debuggability issues are not applicable here as the tests all pass in the default setup.

The majority of test discrepancies would lead to deployment safety violations—the test that passes with the emulator would fail when running with the cloud service (i.e., passing the test provides no safety guarantee on the cloud). For example, a test `CreateTaskHub` in DurableTask uses the Azure Blob API, `Container_Create`, to re-create a previously deleted blob container. This test fails when run with the cloud service with an error message"*the specified container is being deleted; try operation later,*" due to `DurableTaskStorageException` because container deletion is asynchronous and provides no guarantee for the time to finish. However, this test always passes when running with the Azurite emulator, as Azurite always deletes the container synchronously before the API returns.

False alarms are relatively less common than deployment safety issues (Table 4.3). Two (out of four) false alarms are caused by brittle assertions on the error messages returned by the API calls (which are discrepant between the emulator and the cloud service). Such discrepancies can be addressed by enforcing the consistencies of the error messages. One false alarm is caused by a flaky test [44, 45]; the non-deterministic flaky behavior only manifests when running with the emulator, not with the cloud service, due to order differences caused by discrepant timing of API calls. We fixed the flaky tests by adding `await` to enforce the order. The last false alarm is caused by resource discrepancy—the `stress-test` in Orleans exhausted the socket limit of LocalStack (which passes with the cloud service). Such resource discrepancies are essential, and stress tests should not use emulators in the first place.

Key Finding: Deployment safety violations are the major implications of discrepant tests, while false alarms also appear in testing results. Tests of cloud-based software projects need to carefully decide to run on emulators versus cloud services.

Figure 4.5: Key findings of the implications of discrepancies in real-world test suites.

CHAPTER 5: ROOT CAUSE ANALYSIS

To unravel the root causes of the 98 discrepancies discussed in Chapter 4, we debugged every discrepancy by inspecting source code of the emulators (both Azurite and Localstack are open-sourced) and analyzing runtime behavior of the cloud services.

We discuss the discrepancies from the specification perspective. From a high level, both the emulator and the cloud services are implementations of the API specification. So, discrepancies are the result of either incomplete specification or implementation defects. Based on the existing API specifications (Chapter 2), we categorize the discrepancies into:

1. **Defects in existing API specification**: Both Azurite and Localstack adhere to the API specifications of the cloud services (see Chapter 2.1), which specifies types and value constraints of API method parameters and error code. However, the API specifications are incomplete, leading to discrepant validity checks and error responses.

2. **Unspecified behavior that is not considered in existing specifications**: Many discrepancies related to discrepant runtime behavior that is unspecified by, and likely out of the scope of, existing API specifications. A common pattern of unspecified behavior is asynchrony of API effects.

3. **Implementation defects in the emulators or the cloud services**: We also find discrepancies due to defects in the emulator or in the cloud services such as unimplemented components and bugs.

Table 5.1 shows the three categories of discrepancy root causes.

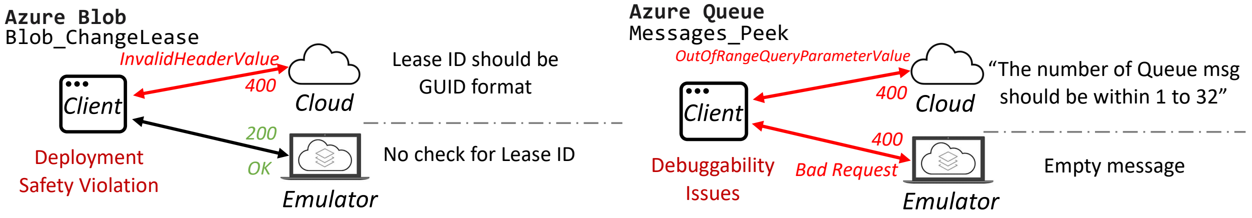
During the project, we detected ten bugs in the two studied emulators, of which six have been confirmed (and five fixed). We also detected two bugs in the cloud backend implementations, which have been reported to the cloud service providers.

5.1 DEFECTS IN EXISTING SPECIFICATIONS

As discussed in Chapter 2, existing cloud service API specification focuses on parameter value constraints and error codes and messages, from which emulators automatically generate stub code that adheres to the specifications (e.g., using AutoRest [32]). However, we still find that a significant percentage of discrepancies are caused by inconsistent validity checks of parameter values as well as inconsistent error code and messages. The reason is incomplete specifications. Table 5.1 shows that incomplete specifications can cause up to 58.1% of the discovered discrepancies in a service.

Table 5.1: Root causes of observed discrepancies.

Service	**Defects in Spec.**	**Unspecified**	**Defects in Impl.**
Azure Blob	18 (58.1%)	1 (3.2%)	12 (38.7%)
Azure Queue	1 (50.0%)	1 (50.0%)	0 (0.0%)
Azure Table	0 (0.0%)	1 (100.0%)	0 (0.0%)
AWS S3	11 (29.7%)	14 (37.8%)	12 (32.4%)
AWS DynamoDB	2 (7.4%)	4 (14.8%)	21 (77.8%)



(a) Parameter value constraints

(b) Error response

Figure 5.1: Discrepancies caused by deficient specifications

Parameter value constraint. Ideally, the API specification should define *all* the value constraints of every input parameter. In reality, API specifications are deficient. We observe discrepant value constraint checks in twelve out of 34 discrepant Azure Storage APIs and ten out of 33 AWS S3 APIs. We observe no such discrepancy in DynamoDB.

Figure 5.1a shows such an example from Azure Blob, where the value of the parameter `x-ms-proposed-lease-id` of the `Blob_ChangeLease` API should be in the GUID format [46]. The Blob service implements a format check, while Azurite does not. As a result, an invalid API call of `Blob_ChangeLease` will be returned successfully by the emulator but rejected by the cloud. However, such value constraint is not specified in the OpenAPI specification of Azure Blob services.

In another common discrepancy case across Azure and AWS, cloud service APIs require authorization to private resources or sensitive operations (e.g., security configuration like `PutBucketAcl`). In its absence, these requests are denied by the cloud service. However, our results revealed that emulators often overlook this constraint, accepting such requests with a 200 OK response, resulting in 8% of discrepancies. We find that the requirement of authentication is commonly included in the specification, but only in text descriptions which is not machine-checkable. Hence, it is not enforced by auto-generated stub code from specifications. In fact, our experience of examining Azure and AWS OpenAPI specifications shows that text-based API descriptions often includes constraints that are not in machine-checkable specifications.

Error response. We also find that specifications can be deficient in comprehensively defining the expected error code and messages and fail to associate them with the APIs, leaving emulator developers to interpret discrepant error messages. Figure 5.1b shows such an example. When a request is made with an out-of-range value for the `numofmessages` parameter, the Azure Queue service provides a detailed message pinpointing the error. In contrast, Azurite only responds with a “Bad Request” error code, offering no specific guidance and impeding debuggability.

Discrepant error responses were particularly prominent in Azure Storage APIs, accounting for 21% (7 out of 34) of the total discrepancies. When we examined the Azure API specifications, we found that the error codes were not associated with the APIs but were defined in a separate list. Differently, we found that AWS specifications have a more structured approach to error code definitions, which were also part of the related API definitions. The latter directly translates to the emulator code. In DynamoDB API specifications, we found structured definitions of 31 unique error codes, including their error messages, exception flags, and documentation. Hence, discrepant error responses are rare in DynamoDB and S3.

Investigating techniques for comprehending and enforcing machine-checkable API specifications may potentially close the gaps and reduce discrepancies, such as specification mining from code and documents [47, 48, 49], especially leveraging recent advances of large language models [50, 51].

Key Finding: The completeness of machine-checkable specifications is still a fundamental challenge, even for simple specifications such as parameter value constraints and error code. Without an effective way towards comprehensive specifications, we expect such discrepancies to remain prevalent.

Figure 5.2: Key findings of discrepancies rooted in incomplete API specifications.

5.2 UNSPECIFIED BEHAVIOR

A few discrepancies were caused by API behavior out of the scope of the existing specification and thus is unspecified. We observed two patterns of unspecified-behavior discrepancies.

We mentioned the first pattern in Chapter 4.3—whether an API is synchronous or asynchronous. For example, Azure Blob’s API `Container_Delete`, which deletes container resources in cloud services, is an asynchronous API. For efficiency consideration, the deletion is not guaranteed to finish before the API returns. Instead, the time to finish the deletion depends on the amount of resources to be deleted. Conversely, emulators always finish

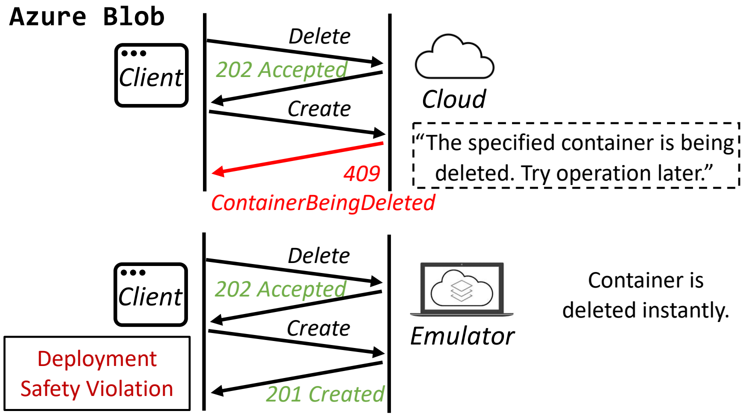


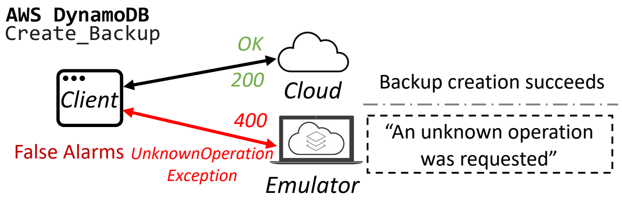
Figure 5.3: Discrepancies caused by unspecified behavior

deletions before returning the API calls. Figure 5.3 depicts such discrepancies. The result is that API sequences involving creating a container, deleting it, and then attempting to recreate it with the same name yielded different results: the cloud service returned 409 `ContainerBeingDeleted`, while the emulator allowed immediate container recreation with 201 `Created`. This pattern also appears in sequences following a deletion API call: the emulator would return a 404 `Not Found` after deletion, while the cloud, busy doing the deletion, would non-deterministically (depending on timing) issue a success response or a 409 Conflict message, "*The specified container is being deleted. Try operation later.*"

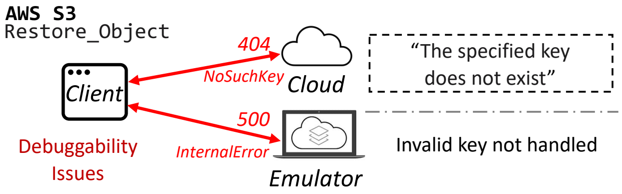
The second pattern is unspecified API behavior on null references (e.g., non-existent objects). For example, the LocalStack emulated S3 APIs used for fetching bucket configuration (e.g., `GetBucketMetricsConfiguration`) or policy (e.g., `GetBucketPolicyStatus`) would return a 200 OK response with an empty policy configuration in the response, when configurations were never set. In contrast, the real S3 APIs returned a 404 error, indicating the configuration was not found. A similar example is APIs used for deleting configurations (e.g., `DeleteBucketMetricsConfiguration`) or object tags (e.g., `DeleteBucketPolicy`). If a configuration was not created, the emulator returned 204 success upon deletion, while the cloud service returned a 404 error. For S3 on LocalStack, 11 discrepancies were caused by this. Such undefined behavior resembles null pointers, a common source of undefined behavior [52].

Key Finding: Two patterns of undefined behavior contribute to discrepancies between emulators and cloud services: (1) the synchrony of the API and (2) null references. Such behavior is currently not considered in cloud service API specification languages and thus not enforced in implementations.

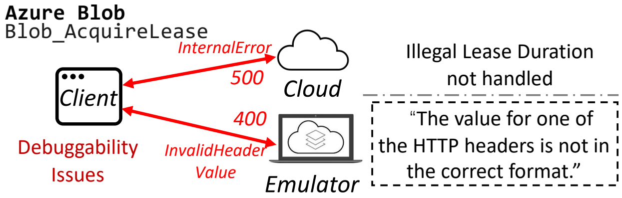
Figure 5.4: Key findings of discrepancies caused by unspecified behavior.



(a) Unimplemented feature.



(b) Bugs in the emulator.



(c) Bugs in the cloud service.

Figure 5.5: Discrepancies caused by implementation defects.

5.3 IMPLEMENTATION DEFECTS

Lastly, we observe discrepancies caused by implementation defects, including unimplemented features and implementation bugs in the emulators and the cloud services.

Unimplemented features in emulators. A significant percentage of discrepancies are due to unimplemented features in emulators, accounting for 18% in Azure Storage, 16% in S3, and 74% in AWS DynamoDB. The emulators' responses to these unimplemented APIs vary. For example, Azurite responds with a 500 error and the message "*Current API is not implemented yet,*" which leads to four wasted retries by the SDK. Whereas the AWS emulator issues a 400 error, as shown in Figure 5.5a, without triggering retries on the client side. According to the coverage reference of LocalStack [31], there are 16 (30%) unimplemented DynamoDB APIs and seven (7%) unimplemented S3 APIs. Azurite states that more features will be supported based on the needs of customers [53].

It is expensive to implement and maintain the large number of cloud service APIs (with high fidelity) in the emulator. Hence, existing emulators take a utility-driven approach to only support commonly used APIs (Figure 4.3). However, if a project relies on unimplemented APIs, the limited support becomes an obstacle for emulator-based testing.

Emulator bugs. Bugs were identified as the root causes of 15% of Azure Storage, 16% of S3, and 4% of DynamoDB discrepancies. Ten emulator bugs were found across the three classes of services (three in Storage, six in S3, and one in DynamoDB). For example, in Figure 5.5b, during tests involving AWS S3's object restoration API, we encountered different responses to invalid keys. While the cloud service correctly rejected invalid keys with a 404 NoSuchKey error, the emulator returned a 500 Internal Error due to a bug that attempted to access a non-existent "storage_class" attribute.

Cloud service bugs. We also find two bugs in the cloud service that resulted in inconsistencies with the emulator. As shown in Figure 5.5c, specifying a lease duration for a blob outside the documented range of 15 to 60 seconds, particularly with an excessively large value, led to a 500 Internal Server Error in the cloud service. Contrastingly, the emulator appropriately responded with a 400 InvalidHeaderValue error, correctly identifying the lease duration as invalid. A similar bug is found in the API for acquiring a container lease.

Key Finding: With the current development of cloud emulators as a reactive practice to cloud services, discrepancies due to unimplemented features and bugs would largely continue, even with active bug fixing and feature requests. Resolving discrepancies needs novel mitigation techniques.

Figure 5.6: Key findings of discrepancies caused by implementation defects.

CHAPTER 6: DISCREPANCY MITIGATION: A DISCUSSION

It is easier to ask for more specifications (Chapter 5.1 and Chapter 5.2) and faster bug fixes (Chapter 5.3). However, it is harder to fundamentally eliminate all the aforementioned discrepancies, as many of them are rooted in the essential complexity of software engineering as well as today's common practices. We discuss a few arguably radical ideas or new practices, hopefully to shed light on viable directions to addressing discrepancies.

6.1 AN ACTIVE ROLE OF CLOUD SERVICE PROVIDERS

We argue that cloud providers have strong incentives to address discrepancies between cloud services and emulators. High-fidelity service emulation would improve developer experience and help promote adoption of cloud services, which is likely the reason providers offer official emulators. Such benefits outweighs the strategy of forcing customers to run all their tests with the cloud services, at least in the long term.

Our fuzzing-based differential testing shows the effectiveness of detecting discrepancies between cloud services and emulators. Cloud service providers can adopt similar practices; they can run the differential testing continuously upon code changes of the emulators or cloud service implementations. Note that cloud providers already run REST API fuzzing to find security bugs in cloud backend implementations (Chapter 9). As service providers have more resources and insights into the implementations, they are in a better position than researchers or cloud-based software developers to discover discrepancies and should take an active role in communicating them.

6.2 FORMAL MODELS AS EMULATORS

Essentially, discrepancies are introduced through the current practice of implementing emulators. Our private communication with a major cloud service provider tells us that the emulators are often not developed by the same engineering team that developed the cloud services, and the emulators are developed *reactive* to the cloud services; for third-party emulators like LocalStack, it is unavoidable. So, without comprehensive formal specifications, discrepancies are inevitable.

One way to resolve discrepancies is to change how emulators are built today. We envision the use of *executable* formal models of cloud services as the emulators. Essentially, the emulator, as the formal model, defines the specifications of the cloud service implementations, which are rigorously tested or verified for compliance. Recent efforts from Amazon [54] show the promise of developing executable reference models as specifications to be checked against the implementation of ShardStore, a key-value storage node of Amazon S3. Similar efforts have been made for file systems [10] and network protocols [55]. In principle, these models can further be developed into first-class emulators for application testing.

6.3 "POSIX" FOR CLOUD SERVICE APIS

One fact that makes cloud emulators particularly prone to discrepancies is the lack of a standard such as POSIX for operating system call APIs. Today, cloud service providers expose APIs with different semantics, constraints, and error codes, even for the same types of services. Without a standardized API, cloud and emulator developers must navigate diverse semantics and error handling for even similar services within the same or across different platforms. As a result, implementations of the APIs, whether by the cloud services or emulators, tend to be error-prone and inconsistent. As a result, we believe that a unified API standard would effectively reduce discrepancies in practice. With the incentives from sky computing [56, 57] and hybrid cloud [58, 59], such a unified API standard may be possible.

6.4 ECONOMIC CLOUD SERVICES FOR TESTING

With the prevalent discrepancies (Chapter 4), testing of cloud-based software would have to largely rely on cloud services. To reduce cost, one can minimize the frequency of running tests with real cloud services (e.g., only do so before deployment, not for CI). If cost is the main concern (a recent survey [60] shows that cost is a major barrier to cloud service adoption), one solution is to provide cheap cloud services for testing. The high cost of cloud services is often driven by pursuits for performance using powerful hardware and fault tolerance using redundancy (Chapter 2.1). But, functional and correctness testing may not need either of them. We envision low-cost cloud services specified for software testing (not for production), with ideas such as using dated hardware [61] and cheap, renewable energy for intermittent services [62]. Certainly, low-cost services do not address other needs of emulators, such as convenience and hermetic environments [63].

6.5 HYBRID CLOUD-EMULATOR TESTING

One principle to mitigate discrepancies without exclusively using cloud services for testing is to acknowledge imperfect emulators and make the best use of them—selectively running tests on emulators when the emulation is not discrepant and on cloud services otherwise. We term such an approach *hybrid cloud-emulator testing* and explore it in Chapter 7.

CHAPTER 7: HYBRID CLOUD-EMULATOR TESTING

To evaluate the effectiveness of hybrid cloud-emulator testing as a short-term discrepancy mitigation (Chapter 6.5), we developed a tool named Et that determines whether a test should be run with emulators or cloud services. The principle is to run discrepant tests with cloud services for safety while running the remaining tests with the emulators for efficiency. Note that it is hard to selectively use emulators and cloud services within a test without expensive state synchronization.

7.1 POLICIES

Et supports three different but complementary policies: depending on if discrepant APIs are known apriori — cloud emulators are expected to communicate fidelity of emulated APIs, though current practice does not provide accurate results (see Chapter 4.1).

7.1.1 Selection by discrepant APIs

This policy assumes apriori knowledge of discrepant APIs. It first runs all tests on the emulator and monitors their REST API calls using a local proxy, as in [64]. If a test invokes a discrepant API, its result is discarded, and the test is rerun on cloud service. Et monitors REST API calls using a local proxy as in [64]. with cloud services; it runs tests that do not invoke any discrepant APIs on the emulator. We use dynamic analysis to record REST APIs invoked by each test by monitoring the requests at the emulator. Static program analysis can also be used. In a CI/CD environment, the dynamic analysis needs to be conducted upon changes to the test and the code under test.

7.1.2 Selection with in-situ API monitoring

Policy §7.1.1 assumes having accurate, comprehensive discrepancy information apriori, which is often not available in practice (as shown in Chapter 4.1, discrepancies are found in developer-certified APIs). Without knowledge of discrepancies, all the tests have to run with cloud services. Moreover, an API can exhibit discrepant behavior only for certain input parameters; hence, statically labeling it discrepant can be too conservative for a test that uses the API with “safe” parameters.

Et supports a new policy that offloads certain tests from the cloud services to the emulators to reduce cost. The high-level idea is to maintain a “safe list” database of API call sequence,

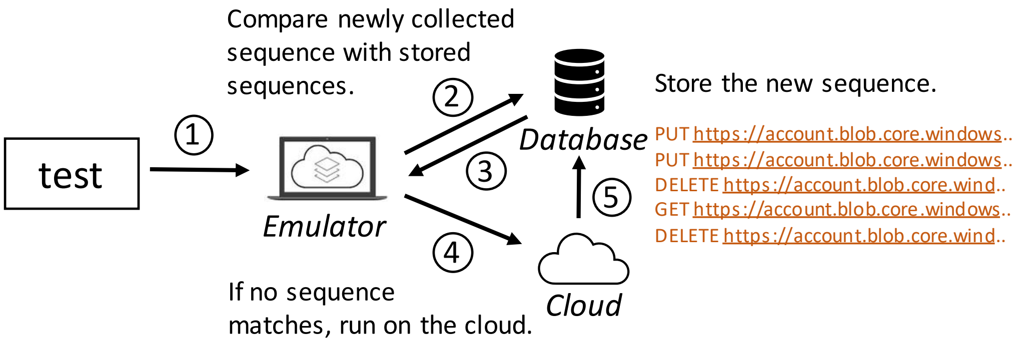


Figure 7.1: Workflow of the policy §7.1.2 of Et.

including each call's request (with parameters) and response. Et starts with an empty safe list. For each test, it first runs the test on the emulator and monitors the API call sequences. If the sequence is not present in the safe list, the test is assumed to be discrepant and is rerun on the cloud service. The API sequence is added to the safe list if its result from the emulator matches that from the cloud service. On the other hand, if the API sequence is found in the safe list, then Et skips running the test with the cloud services, because the fidelity of interactions has already been validated by a real cloud-based test run. Figure 7.1 shows the workflow.

Note that the analysis considers the entire API call sequence instead of individual APIs (Figure 5.3 shows an example where the discrepancy only manifests with specific sequence). Et serializes the API calls at its local proxy. For fast comparison, each sequence is stored as an ordered list of hashes while each hash represents an API along with its parameters and response; collisions are chained upon occurrence. We implement masks to exclude intrinsically non-deterministic parameters and fields in the response, such as timestamps.

Note that this policy accounts for non-determinism of test execution due to multi-threading and event-based asynchrony. The essence of the policy is to validate the external behavior of API calls issued by the test on the emulator with the real cloud services. Et does not make assumptions on the internal implementation of test code or system under test.

Despite no apriori discrepancy knowledge, this policy can outperform the policy §7.1.1 in certain cases, because it performs a fine-grained, parameter-level analysis. As shown in Chapter 4.3, discrepancies typically manifest via specific parameter values rather than universally across the API; a discrepant test can still be run on the emulator if it does not manifest discrepant results.

Table 7.1: Savings of cloud service API invocations with different policies of ET (§7.1.1, §7.1.2, and §7.1.3). The numbers are averaged across five commits.

Project	**Total Tests**	**Total Requests**	**# Saved Requests**		
			§7.1.1	**§7.1.2**	**§7.1.3**
Orleans	189	117,905	29.4%	0.4%	29.6%
Insights	171	5,249	4.3%	47.2%	47.2%
Durabletask	101	79,654	0%	0%	0%
Streamstone	75	590	0%	99.2%	99.2%
IdentityAzureTable	51	9,860	100%	0.4%	100%

7.1.3 Combined policy.

We support a combined policy that integrates Policy §7.1.1 and 7.1.2—only conduct in-situ discrepant analysis for discrepant tests. Basically, if we know a test is not discrepant apriori, we always run it on the emulator.

7.1.4 Evaluation

We evaluate the three policies described in Chapter 7.1 in terms of cost savings measured by the number of calls to cloud APIs. We assume a continuous integration (CI) setup as Policy §7.1.2 only benefits continuous testing, and its benefit is correlated with the comprehensiveness of the safe list.

We select five projects that use Azure APIs (ET currently only supports .NET applications) and use all the related tests (Table 3.2). We select the five projects with the most tests that invoke Azure APIs. We evaluate ET with the most recent five commits to simulate CI and record the cost saving for each commit (we run only five commits due to the constraint of our cloud education credits). All the tests will be run for each commit that changes system code or test code. Note that regression test selection [65, 66, 67] does not apply to those tests which are not unit tests but mostly integration and system tests.

Results. As shown in Table 7.1, ET effectively reduces the amount of invocations to cloud APIs. Interestingly, the three policies bring different benefits across projects, with the combined policy §7.1.3 achieving the most cost savings.

Policy §7.1.1 achieves substantial savings for two out of five projects. Specifically, it achieves a 100% saving for IdentityAzureTable where none of its tests issues a discrepant API. However, it achieves 0 saving for Streamstone and DurableTask, because all their tests issue at least one discrepant API.

Policy §7.1.2 achieves substantial savings for two different projects (Insights and Streamstone) but not the others. Our investigation reveals that the effectiveness of Policy §7.1.2 largely depends on the ordering determinism of API call sequences. Since tests that invoke cloud APIs are typically large system/integration tests with large numbers of API calls, the sequences recorded during the tests on five commits are insufficient. We expect that a longer continuous testing process may increase the benefit, which remains our future work.

Key Finding: Et shows that by selectively running tests on emulators, it is promising to reduce the cost of cloud-based software testing, in terms of the cost of calling cloud service APIs, while achieving high-fidelity testing.

Figure 7.2: Key findings of Et

CHAPTER 8: THREATS TO VALIDITY

Our study is based on the five cloud services (three Azure services and two AWS services) and two emulators (Azurite and LocalStack). We believe that the studied cloud services and emulators are representative, but our results may not generalize to other cloud services, especially those using different practices (e.g., for specifications).

Similarly, our analysis of discrepancy impacts (Chapter 4.3) and Et's evaluation results (Chapter 7.1.4) are based on existing test suites of a few cloud-based projects. They may not generalize to other projects as they depend on API usage characteristics of projects and their tests (e.g., invoked cloud APIs and their frequencies). In principle, projects that use cloud services more extensively face higher impacts caused by discrepancies. As cloud-based programming models are more prevalent, driven by recent trends like serverless [68, 69], sky computing [56, 57], and hybrid cloud [58, 59], we expect more extensive integration of cloud services in modern software projects. Hence, we expect cloud-emulator discrepancies to be common issues for software testing.

The discrepancies analyzed in this thesis are limited to the black-box SDK API fuzzer we developed based on RESTler (Chapter 3), and we do not claim completeness of studied discrepancies. A more powerful fuzzer, especially a white-box one, may cover more discrepant APIs. As a best effort, we run our fuzzer for more than ten hours against each studied emulator and stop the fuzzing when we do not observe any new discrepancies.

Lastly, we are not concerned with faults that occur during the API invocations, such as timeout due to network delays. Recent work [64] shows that timeouts on both the request and response paths of a REST API invocation can reveal different behaviors, which we would like to study as future work.

CHAPTER 9: RELATED WORK

REST API fuzzing. Recent work has developed advanced REST API fuzzing techniques to test web and cloud services, with the goal of finding bugs and vulnerabilities in web service implementations [5, 35, 36, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79]. Differently, this thesis focuses on the software projects that use cloud services, instead of the backend implementations of cloud services. Our goal is to understand discrepancies between the emulator and the cloud services and their implications for software testing.

We developed our fuzzer based on the fuzzing approach of RESTler [35]. As discussed in Chapter 3, we did not directly use RESTler (or other REST API fuzzers) because most projects only interact with SDK APIs, not REST APIs, and REST API fuzzers generate API calls that would not be output by SDKs.

Fidelity of emulation. Prior work has studied the fidelity of emulation environments in other domains, such as honeypots for security analysis [80, 81, 82]. The closest related work is the research on the fidelity of emulated execution environments such as virtual devices for mobile app testing [83, 84]. The goal is to maximize app testing on emulated devices and minimize testing on real physical mobile devices (which are more expensive and hard to manage [83]).

Our work shares similar high-level goals and tradeoffs (cost-efficiency versus safety). But, we address a different fidelity problem raised by the emerging cloud-based programming model. The discrepancies are not due to deficiency or incompleteness of device emulation but are rooted in inconsistent implementations of weakly specified APIs.

Backward compatibility. The studied discrepancies are different from backward incompatibility studied in prior work [5, 85, 86, 87, 88]. We do not study the evolution of emulators or cloud service APIs in this thesis, though certain discrepancies can be caused by regression [5]. There are also studies on mock libraries for unit tests [89, 90, 91]; few of them concern fidelity of mock objects—unlike emulation, mocking is not expected to provide fidelity but offers a way to control external APIs.

CHAPTER 10: CONCLUSION

With the rise of cloud-based, serverless programming models, testing cloud-based software safely and cost-efficiently becomes a challenge. This thesis provides a comprehensive examination of the fidelity of cloud emulators in the context of software development and local testing against real cloud services. Through systematic analysis, we have identified significant discrepancies in emulators —37% of APIs (94 out of 255) from Azure and AWS cloud services exhibited discrepant behaviors, which potentially impact the safety and trustworthiness of testing results of cloud-based software. We discuss the accidental and essential root causes of these discrepancies and effective strategies for mitigation.

Our findings emphasize the need for more standardized and comprehensive cloud API specifications, which form the foundation for emulators. As a short-term solution, the proposed hybrid cloud-emulator testing framework has been effective. It provides accurate testing outcomes at lower costs compared to using the actual cloud. While this tool addresses many of the current challenges, future work should explore the additional solutions proposed in this thesis, which may offer greater cost-effectiveness and better alignment between emulator and cloud behaviors.

As cloud services continue to evolve, the insights from this thesis will guide future efforts to improve cloud emulation practices. This work not only encourages ongoing efforts but also calls for increased collaboration between cloud service providers and emulator developers. By working together towards the goal of high-fidelity emulators, they can achieve more. Ultimately, this research contributes to enhanced local developer testing by providing a clearer path toward reliable cloud emulators in cloud-based software development.

REFERENCES

[1] "Azure products," https://azure.microsoft.com/en-us/products, 2024.

[2] "Google Cloud products," https://cloud.google.com/products, 2024.

[3] "AWS Cloud Products," https://aws.amazon.com/products, 2024.

[4] NuGet, "WindowsAzure.Storage NuGet," https://www.nuget.org/packages/WindowsAzure.Storage, 2024.

[5] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential Regression Resting for REST APIs," in *ISSTA*, 2020.

[6] "Azure Blob Storage Cost," https://azure.microsoft.com/en-us/pricing/details/storage/blobs/, 2023.

[7] "AWS Pricing Calculator," https://calculator.aws/#/?nc2=pr&refid=f42fef03-b1e6-4841-b001-c44b4eccaf41, 2024.

[8] John Micco, "The State of Continuous Integration Testing @Google," in *ICST*, 2017.

[9] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing Transition-Based Test Selection Algorithms at Google," in *ICSE-SEIP*, 2019.

[10] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell, "SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems," in *SOSP*, 2015.

[11] Rick Timmis, "Xiatech accelerates their development workflows on cloud using LocalStack!" https://localstack.cloud/blog/2023-07-05-case-study-xiatech/.

[12] Microsoft Docs, "Use the Azurite emulator for local Azure Storage development," https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azurite, 2024.

[13] "Introduction to Azure Storage," https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction, 2024.

[14] azurite-1465, "Table storage having wrong constraint," https://github.com/Azure/Azurite/issues/1465.

[15] azurite-946, "Missing parameter in API response," https://github.com/Azure/Azurite/issues/946.

[16] azurite-5, "Requesting for a feature," https://github.com/Azure/Azurite/issues/5.

[17] nuget-insights-30, "List of issues which blocked CI tests for Insights," https://github.com/NuGet/Insights/issues/30.

[18] Localstack, “LocalStack – A fully functional local cloud stack,” https://localstack.cloud/, 2024.

[19] Azure/durabletask, https://github.com/Azure/durabletask.

[20] “AWS S3 REST APIs,” https://docs.aws.amazon.com/AmazonS3/latest/API/API_Operations.html, 2023.

[21] “Azurite Swagger of Azure Blob Storage,” https://github.com/Azure/Azurite/blob/main/swagger/blob-storage-2021-10-04.json.

[22] “AWS S3 Python SDK APIs,” https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html, 2023.

[23] “Azure SDK for Python,” https://azuresdkdocs.blob.core.windows.net/$web/python/azure-storage-blob/12.0.0b5/index.html#clients, 2024.

[24] “OpenAPI Specification,” https://www.openapis.org/, 2024.

[25] “Azure Blob Storage Cost Breakdown,” https://azure.github.io/Storage/docs/application-and-user-data/code-samples/estimate-block-blob/, 2023.

[26] “Azure Locally-redundant storage (LRS),” https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy#locally-redundant-storage, 2024.

[27] “Azure Zone-redundant storage (ZRS),” https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy#zone-redundant-storage, 2024.

[28] “Azure Locally-redundant storage (GRS),” https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy#geo-redundant-storage, 2024.

[29] Brian Harry, “The largest Git repo on the planet,” https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/, 2017.

[30] R. Potvin and J. Levenberg, “Why Google Stores Billions of Lines of Code in a Single Repository,” *Communications of the ACM (CACM)*, vol. 59, no. 7, pp. 78–87, June 2016.

[31] “LocalStack Coverage,” https://docs.localstack.cloud/references/coverage/, 2023.

[32] Azure/autorest, https://github.com/Azure/autorest, 2024.

[33] “Regeneration Protocol Layer from Swagger by Autorest,” https://github.com/Azure/Azurite/blob/main/CONTRIBUTION.md#regeneration-protocol-layer-from-swagger-by-autorest, 2024.

[34] “LocalStack Weekly ASF Update Workflow,” https://github.com/localstack/localstack/blob/master/.github/workflows/asf-updates.yml, 2024.

[35] V. Atlidakis, P. Godefroid, and M. Polishchuk, “RESTler: Stateful REST API Fuzzing,” in *ICSE*, 2019.

[36] V. Atlidakis, P. Godefroid, and M. Polishchuk, “Checking Security Properties of Cloud Service REST APIs,” in *ICST*, 2020.

[37] LocalStack, “Coverage S3 - LocalStack Documentation,” https://docs.localstack.cloud/references/coverage/coverage_s3/, 2023.

[38] LocalStack, “Coverage DynamoDB - LocalStack Documentation,” https://docs.localstack.cloud/references/coverage/coverage_dynamodb/, 2023.

[39] LocalStack, “Parity Testing - LocalStack Documentation,” https://docs.localstack.cloud/contributing/parity-testing/, 2023.

[40] “Azurite Testing,” https://github.com/Azure/Azurite/blob/main/CONTRIBUTION.md#testing.

[41] “Azurite Support Matrix,” https://github.com/Azure/Azurite?tab=readme-ov-file#support-matrix.

[42] “Azure Storage API: ContainerRestore,” https://github.com/Azure/Azurite/blob/d544d16f910e490fdd9db5565459df701895308f/swagger/blob-storage-2021-10-04.json#L1554, 2024.

[43] “Azure Storage API: BlockBlob-StageBlockFromURL,” https://github.com/Azure/Azurite/blob/d544d16f910e490fdd9db5565459df701895308f/swagger/blob-storage-2021-10-04.json#L7069, 2024.

[44] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An Empirical Analysis of Flaky Tests,” in *FSE*, 2014.

[45] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, “A Survey of Flaky Tests,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 17, pp. 1–74, Oct. 2021.

[46] “Lease Blob - Microsoft Documentation,” https://learn.microsoft.com/en-us/rest/api/storageservices/lease-blob?tabs=microsoft-entra-id.

[47] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, “C2S: Translating Natural Language Comments to Formal Program Specifications,” in *ESEC/FSE*, 2020.

[48] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, “DocTer: Documentation Guided Fuzzing for Testing Deep Learning API Functions,” in *ISSTA*, 2022.

[49] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, “Analyzing APIs Documentation and Code to Detect Directive Defects,” in *ICSE*, 2017.

[50] D. Xie, B. Yoo, N. Jiang, M. Kim, L. Tan, X. Zhang, and J. S. Lee, “Impact of Large Language Models on Generating Software Specifications,” *arXiv:2306.03324*, 2023.

[51] S. Mandal, A. Chethan, V. Janfaza, S. M. F. Mahmud, T. A. Anderson, J. Turek, J. J. Tithi, and A. Muzahid, “Large Language Models Based Automatic Synthesis of Software Specifications,” *arXiv:2304.09181*, 2023.

[52] C. Hathhorn, C. Ellison, and G. Roşu, “Defining the Undefinedness of C,” in *PLDI*, 2015.

[53] “Azurite README,” https://github.com/Azure/Azurite/blob/main/README.md.

[54] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang, “Specifying and Checking File System Crash-Consistency Models,” in *ASPLOS*, 2016.

[55] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, “Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets,” in *SIGCOMM*, 2005.

[56] S. Chasins, A. Cheung, N. Crooks, A. Ghodsi, K. Goldberg, J. E. Gonzalez, J. M. Hellerstein, M. I. Jordan, A. D. Joseph, M. W. Mahoney, A. Parameswaran, D. Patterson, R. A. Popa, K. Sen, S. Shenker, D. Song, and I. Stoica, “The Sky Above the Clouds: A Berkeley View on the Future of Cloud Computing,” *arXiv:2205.07147*, 2022.

[57] I. Stoica and S. Shenker, “From Cloud Computing to Sky Computing,” in *HotOS*, 2021.

[58] Google Cloud, “What is a Hybrid Cloud?” https://cloud.google.com/learn/what-is-hybrid-cloud.

[59] IBM Hybrid Cloud, “Hybrid cloud solutions,” https://www.ibm.com/hybrid-cloud.

[60] M. Loukides, “The Cloud in 2021: Adoption Continues,” O’Reilly Medial, Tech. Rep., 2021.

[61] J. Wang, U. Gupta, and A. Sriraman, “Giving Old Servers New Life at Hyperscale,” in *HotInfra*, 2023.

[62] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini, “Providing SLOs for Resource-Harvesting VMs in Cloud Platforms,” in *OSDI*, 2020.

[63] C. Narla and D. Salas, “Hermetic Servers,” https://testing.googleblog.com/2012/10/hermetic-servers.html, Oct. 2012.

[64] Y. Chen, X. Sun, S. Nath, Z. Yang, and T. Xu, “Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker,” in *NSDI*, 2023.

[65] G. Rothermel and M. Harrold, “Analyzing Regression Test Selection Techniques,” *IEEE Transactions on Software Engineering (TSE)*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[66] G. Rothermel and M. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, p. 173–210, Apr. 1997.

[67] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation: A Survey," *Software Testing, Verification & Reliability*, vol. 22, no. 2, p. 67–120, Mar. 2012.

[68] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. M. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," University of California at Berkeley, Tech. Rep. UCB/EECS-2019-3, 2019.

[69] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the Planet of Serverless Computing: A Systematic Review," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 32, no. 5, pp. 1–61, July 2023.

[70] C. Lyu, J. Xu, S. Ji, X. Zhang, Q. Wang, B. Zhao, G. Pan, W. Cao, P. Chen, and R. Beyah, "MINER: A Hybrid Data-Driven Approach for REST API Fuzzing," in *USENIX Security*, 2023.

[71] P. Godefroid, B.-Y. Huang, and M. Polishchuk, "Intelligent REST API Data Fuzzing," in *ESEC/FSE*, 2020.

[72] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing Program Input Grammars," in *PLDI*, 2017.

[73] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine Learning for Input Fuzzing," in *ASE*, 2017.

[74] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial Testing of RESTful APIs," in *ICSE*, 2022.

[75] J. C. Alonso, "Automated Generation of Realistic Test Inputs for Web APIs," in *ESEC/FSE*, 2021.

[76] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Automated Black-Box Testing of RESTful Web APIs," in *ISSTA*, 2021.

[77] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, Jan. 2019.

[78] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs," in *ICST*, 2020.

[79] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: Model-based RESTful API Testing with Execution Feedback," in *ICSE*, 2022.

[80] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang, "Understanding Fileless Attacks on Linux-based IoT Devices with HoneyCloud," in *MobiSys*, 2019.

[81] C. Kreibich and J. Crowcroft, "Honeycomb – Creating Intrusion Detection Signatures Using Honeypots," in *SIGCOMM-CCR*, 2004.

[82] M. D. Vrable, J. T. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage, "Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm," in *SOSP*, 2005.

[83] H. Lin, J. Qiu, H. Wang, Z. Li, L. Gong, D. Gao, Y. Liu, F. Qian, Z. Zhang, P. Yang, and T. Xu, "Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility," in *MobiCom*, 2023.

[84] H. Cai, Z. Zhang, L. Li, and X. Fu., "A Large-Scale Study of Application Incompatibilities in Android," in *ISSTA*, 2019.

[85] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis," in *ICSE*, 2020.

[86] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and Detecting Software Upgrade Failures in Distributed Systems," in *SOSP*, 2021.

[87] C. Zhu, M. Zhang, X. Wu, X. Xu, and Y. Li, "Client-Specific Upgrade Compatibility Checking via Knowledge-Guided Discovery," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 32, no. 4, pp. 1–31, May 2023.

[88] Y. Zhao, L. Li, K. Liu, and J. C. Grundy, "Towards Automatically Repairing Compatibility Issues in Published Android Apps," in *ICSE*, 2022.

[89] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic Test Factoring for Java," in *ASE*, 2005.

[90] X. Wang, L. Xiao, T. Yu, A. Woepse, and S. Wong, "An Automatic Refactoring Framework For Replacing Test-Production Inheritance by Mocking Mechanism," in *ESEC/FSE*, 2021.

[91] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated Unit Test Generation for Classes with Environment Dependencies," in *ASE*, 2014.