Building Reliable Cloud Services Using Coyote Actors

Pantazis Deligiannis
Microsoft Research, USA
pdeligia@microsoft.com

Narayanan Ganapathy*
Facebook, USA
narg@fb.com

Akash Lal
Microsoft Research, India
akashl@microsoft.com

Shaz Qadeer†
Novi, USA
shaz@fb.com

ABSTRACT

Cloud services must typically be distributed across a large number of machines in order to make use of multiple compute and storage resources. This opens the programmer to several sources of complexity such as concurrency, order of message delivery, lossy network, timeouts and failures, all of which impose a high cognitive burden. This paper presents evidence that technology inspired by formal-methods, delivered as part of a programming framework, can help address these challenges. In particular, we describe the experience of several engineering teams in Microsoft Azure that used the open-source Coyote Actor programming framework to build multiple reliable cloud services. Coyote Actors impose a principled design pattern that allows writing formal specifications alongside production code that can be systematically tested, without deviating from routine engineering practices. Engineering teams that have been using Coyote have reported dramatically increased productivity (in time taken to push new features to production) as well as services that have been running live for months without any issues in features developed and tested with Coyote.

CCS CONCEPTS

• **Software and its engineering → Software verification**; *Cloud computing*; *Software fault tolerance*;

KEYWORDS

Reliability, Cloud Services, Systematic Testing

*This work was done while the author was at Microsoft.
†This work was done while the author was at Microsoft.

SoCC '21, November 1–5, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3486983

ACM Reference Format:
Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. 2021. Building Reliable Cloud Services Using Coyote Actors. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–5, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3486983

1 INTRODUCTION

The public cloud offers access to an easily-managed, pay-on-use model of renting compute and storage resources. Increasingly, many companies are moving their business workloads to the cloud [11, 12]. This requires designing software services that execute on the cloud, making effective use of the available resources. However, developing such services is challenging: service components are spread across multiple virtual machines and data centers, and communication must happen over the network. To build a reliable cloud service, developers must defend against all common pitfalls of distributed systems: the concurrency from multiple executing processes, unreliable networks (e.g., out-of-order delivery, or message loss/duplication), as well as hardware/-software failures. In this paper, we refer to these combined challenges as sources of *non-determinism*. It is no surprise that the presence of such non-determinism leads to bugs in production, causing tangible loss of business and customer trust [2, 38, 40].

The research community has made several attempts at finding distributed-systems bugs, commonly through the use of *systematic testing* tools. Examples include Chess [33, 34], MoDist [44], dBug [42] and SAMC [20]. These tools take over the non-determinism in a test environment and control it to explore many different program executions. Both exhaustive (up to a bound) and random explorations have proven to be very effective at finding bugs. However, despite of this success, there has been no visible change in software development practices followed in the industry. Chances are that the next time around a new system is built, it will be built in the same manner as before, leading to the same kinds of bugs seen in previous systems. Without a change in the software development process, the likely impact of any bug-finding technique will be limited.

This paper presents evidence that the situation is not as grim as outlined above. Tools and techniques developed by the research community can indeed have considerable impact in the industry for engineering distributed systems. We share experience from the adoption of the open-source Coyote programming framework [37] in Microsoft Azure for building production cloud services using the actor style of programming [1].[1] Coyote Actors impose a principled design pattern, allowing the implementation to closely resemble its high-level design. Coyote also provides mechanisms for programmatically expressing non-determinism and writing detailed safety and liveness specifications. Finally, Coyote comes with automated testing capabilities that encapsulate the state-of-the-art in systematic testing. This enables high-coverage testing of the production code against its specifications and provides deterministic reproducibility of bugs. Putting these pieces together, Coyote allows developers to effectively iterate through the *design-implement-test* cycle faster than otherwise, leading to accelerated development.

To illustrate the benefits of using Coyote, we provide a detailed description of *PoolManager*, one of three core components of the Azure Batch service (ABS) [22] that was written from scratch using Coyote actors. ABS is a popular job scheduling and compute management service, offered by Azure, managing over *hundreds of thousands* of VMs on the cloud. ABS allows a user to create a collection of compute nodes and schedule a parallel job across these nodes. PoolManager is responsible for creating and managing the collection of compute nodes (also called a *pool*). A previous version of PoolManager, developed over several years using standard engineering practices, had an outdated design that was unable to manage the increasing demands of Azure Batch. It was hard to maintain and test, making feature addition unacceptably slow. This prompted the ABS team to rewrite the PoolManager, this time adopting Coyote.

The ABS engineers (both junior and senior) were able to move faster and be more confident in their code changes because they could achieve high-coverage testing with Coyote. Writing detailed specifications alongside production code became an integral part of their daily development process. The team reported that the coverage obtained with systematic testing was much higher than even with days of stress testing. Coyote found *hundreds* of bugs that were fixed fast, and often without ever getting checked-in. For a few bugs that we were able to snapshot, it was unlikely that they would have been found using stress testing, or other conventional testing methods, because they required several failures and timeout events to interleave.

The ABS team gained considerable confidence in Coyote testing as the PoolManager development proceeded: once a feature was tested with Coyote, it would *just work* when put into production. The current state of practice in the team is that each code check-in to the *main* branch must clear all available Coyote tests. It was a unique experience for the team to get exhaustive (in reality, high-coverage) testing of their code changes readily available on their desktop as they were developing a distributed system. The debugging process was also significantly improved: each time Coyote found a bug, engineers could deterministically replay the buggy trace, attach a debugger, set breakpoints and step-into the code. The majority of the new PoolManager development took *only six months*, considerably faster than the previous version. For some time, both versions of PoolManager existed simultaneously. During this time, the team had to add a new feature (for supporting low priority preemptible VMs in the pools) to both versions. The addition in the old PoolManager took six person months, whereas the addition in the new Coyote version took *just one person month*. The new PoolManager has now been operating for over a year with *no reported bugs in production* for Coyote tested features. There were occasional bugs, but they all pointed to features outside the scope of their Coyote tests.

The Azure Batch PoolManager is the first production-scale system, to the best of our knowledge, to have been developed along with continuous validation of safety and liveness specifications. The experience of the ABS team was not isolated. Given their success, several other teams in Azure adopted Coyote in their engineering process. Currently, Coyote actors have been used in Azure for building nine production services, with several more services in the planning stage (§6). Furthermore, Coyote has 100% user retention so far: once a team started using it, they have continued to use it for writing new cloud services.

The main contributions of this paper are as follows:

- We present PoolManager, the first production-scale system to have been developed simultaneously with continuous validation of the actual code against its safety and liveness specifications; both design and implementation was done by engineers (not researchers) to meet a specific business need.
- PoolManager is a stateful microservice that requires storing its state reliably so that it can be restored after a failure (known as a *failover*). Getting the failover logic correct is often hard. We give a novel methodology for failover testing (§4).

[1] Coyote supports multiple programming frameworks, including Task-based programming, but this report mainly focusses on the experience with its Actor programming framework. Coyote's Actor framework was earlier called P# [7].

- We discuss the experience of several Azure engineering teams with using Coyote actors, for building highly-reliable cloud services (§6).

The rest of this paper is organized as follows: §2 provides background; §3 outlines the design and implementation of the PoolManager using Coyote; §4 focuses on testing of the PoolManager; §5 lists the improvements made to Coyote actors to support the development of production systems; §6 summarizes the experience of several Azure engineering teams with using Coyote; and finally §7 presents related work.

2 OVERVIEW

2.1 The Azure Batch Service (ABS)

ABS is a popular generic job scheduling service offered by Microsoft Azure [22]. ABS allows a user to execute a parallel job in the cloud. The job can consist of multiple tasks with a given set of dependencies. ABS will execute the tasks in dependency order while attempting to exploit as much parallelism between independent tasks as possible. Unlike distributed schedulers such as Apache Yarn [41] and Mesos [17] that typically require to be installed on a pre-created set of VMs, ABS integrates scheduling with VM management. ABS can *auto-scale* (i.e., spin up and down) the number of created VMs based on the needs of each job as well as a variety of parameters such as CPU, memory and I/O metrics on VMs, and preemption rate.

The high-level architecture of ABS is shown on the left side of Figure 1. Each *region* (i.e., a geographical location that hosts one or more data centers) has a resource provider and several *schedulers*. A breakdown of the resource provider is shown on the right side of Figure 1. The resource provider has a front-end or gateway service that routes requests to back-end managers that support the CRUD operations for specific entities: *user accounts*, *pools* and *jobs*, with relevant data stored in Azure Storage [27] for persistence.

ABS is a multi-tenant service and a user account is the multi-tenant isolation boundary. Each account is associated with a *quota* that limits the amount of compute resources that can be allocated for a scheduled job. All resources used by ABS for executing a job are billed to the corresponding account. After creating and registering an account, a user can create a *pool* that refers to a collection of compute nodes (VMs). The sum total of cores of all VMs across all pools associated with an account must be less than the corresponding core quota limit of the account. The pool can be of fixed size, or set to auto-scale. Once a pool is created, the user can submit a job to schedule on the pool. Each of the account, pool and job managers are multi-instance partitioned services (partitioned by account). The partitions themselves are managed by the partition manager. The design of the partition manager is out of scope for this paper but the various services have to honor partition manager requests to start/stop/split/merge partitions.

PoolManager. At the heart of the ABS functionality is the microservice component called PoolManager. The PoolManager interacts with many other components in the system, such as the job manager, Azure Storage [27] for storage, Virtual Machine Scale Sets (VMSS) [29] for VM management and Azure Subscriptions [28] for billing accounts. ABS needs to respond to auto-scale requirements very rapidly. To achieve this, it must support functionality to cancel outstanding operations so that further changes to resources can be made. The functionality must be provided with low latency, high throughput, high availability and scale. Note that all services that ABS interacts with are publicly available and ABS uses the same APIs that are published externally. This implies that ABS must obey all the rules and limitations enforced by these services. This point was an important design consideration, especially for ABS quota and billing management.

The PoolManager component had to be redesigned for a variety of reasons. The old design split the work of creating pools between the PoolManager and the scheduler. The PoolManager managed pool entities and quotas while the scheduler cached a pool of VMs. Caching of VMs was no longer feasible for ABS: as its usage increased, customers wanted VMs of different sizes and OS images, etc., so it became too costly to hold the VMs in the scheduler. The old design made the scheduler very complicated. It was also harder to dynamically scale up and down the number of schedulers because they were involved in VM management. The goal of the redesign was to move the quota management and the actual VM allocation to a single component (PoolManager) where it can be partitioned and scaled independently of the scheduler component. This helped the scheduler become a very lightweight component that can be spun up and down quickly, as it now only focuses on job scheduling. The redesign also allowed the ABS team to easily incorporate different types of scheduling policies.

The old PoolManager did have some unit and integration tests but the ABS team felt that the tests did not provide much confidence in the overall reliability of the PoolManager. It was important to remedy this situation as well. The PoolManager is a stateful component that operates in a distributed-systems environment. Testing of such components is challenging. For instance, the VM hosting a PoolManager instance may fail or reboot without warning. Operations on pools are long-running asynchronous activities, thus, the developer must anticipate and account for failures that can happen in the middle of such an operation. We refer to the part of a program's design that deals with recovery from

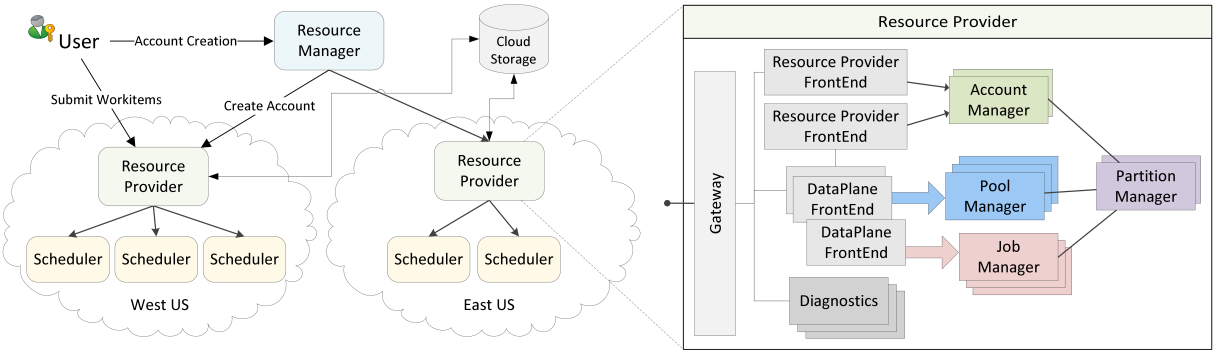

Figure 1: The Azure Batch service architecture on the left, and the Resource Provider architecture on the right.

failures as *failover logic* and testing for its correctness as *failover testing*. Failovers are not the only challenge: one must also correctly deal with issues such as message re-orderings, timeouts and error handling (when interacting with other services). Due to inadequate testing technology, most often, errors arising from these types of issues are discovered late in the development cycle, or even after deployment when they are very costly to debug and fix. Ideally, we should be able to discover and fix these kinds of issues well before the software is deployed in production.

A further requirement of the PoolManager design was that the entire code base should be asynchronous and non-blocking. This is to ensure that PoolManager remains responsive to cancellation requests: the user is allowed to cancel an outstanding operation at any time. In the old design, there were dedicated threads that blocked synchronously and when the process ran out of threads, the system could not process more requests.

2.2 An Introduction to the Coyote Actor Framework

Coyote [37] provides an open-source actor-based [1] .NET programming framework. (As mentioned before, while Coyote supports other programming models as well, we only focus on actors in this paper.) We will refer to a Coyote actor as a `machine`. A program can dynamically create any number of Coyote actors that execute concurrently with each other and communicate via messages called `events`. Each actor is equipped with an inbox where incoming `events` get enqueued, and executes an event-handling loop that waits for `events` to arrive and processes them sequentially one after the other. An actor can internally define a *state machine* structure for programmatic convenience, which happened to be the common use-case for the work described in this paper. For this reason, we will term an actor as a `machine` in this paper.

Coyote actors provides a higher-level concurrency model compared to using threads and locks. A `machine` encapsulates its own state that is not shared with other `machines`, and synchronization is limited to sending `events`. This means that all communication points between `machines` are clearly marked in code.

Coyote is designed in a manner that allows robust testing of non-deterministic systems. To this effect, Coyote requires developers to declare all non-determinism present in their code, after which they can use the Coyote tester to exercise (in the limit) all possible behaviors of a given test case. The tester understands the non-determinism that arises from concurrency between `machines`. It uses hooks into the Coyote runtime to control the scheduling of `machines`. There can be other forms of non-determinism in the code. Coyote exposes an API to generate unconstrained Boolean and integer values. We refer to this API as `RandomBoolean` in the rest of this paper. It is the responsibility of the programmer to *model* the non-determinism in their code using this API. We illustrate this point using an example.

Consider building an application that requires running multiple Coyote `machines` distributed over a network. The developer is interested in testing the implementation against a *lossy* network, as the code must work correctly even if the network arbitrarily drops messages. The developer first writes a test that initializes all Coyote `machines` under a mocked distributed environment so that the code can execute in a single-process setting. This mocked environment can model the network to express its lossy behavior. Figure 2 shows an illustration for such a mock. The application (not shown) is designed against an interface of the network (`INetworkingService`), which is then mocked (`MockNetworkingService`) for testing purposes. The mocked method `SendMessage` calls `RandomBoolean` to decide if it is going to deliver the `event` or not. When it must deliver, it directly addresses the target `machine` and delivers the `event` via a Coyote `SendEvent`. Once external dependencies are substituted with mocks to make the

```
interface INetworkingService {
  void SendMessage(string endPoint, Event msg);
}
class MockNetworkingService : INetworkingService {
  // Map: Endpoint -> Machine hosted at the endpoint
  Dictionary<string, ActorId> machineMap;
  IActorRuntime CoyoteRuntime;

  void SendMessage(string endPoint, Event msg) {
    if(CoyoteRuntime.RandomBoolean()) {
      CoyoteRuntime.SendEvent(machineMap[endpoint], msg);
} } }
```

Figure 2: A mock for a lossy network.

test self-contained (as in standard unit-testing), the Coyote tester repeatedly executes the test, each time exploring a different interleaving of concurrent actions as well as resolving `RandomBoolean` calls with different values.

The Coyote tester supports many state-of-the-art search strategies inspired from the systematic testing literature [5, 10, 31], and makes it easy to add new strategies as the research community comes up with new algorithms. By default, the tool is configured to execute a portfolio of search strategies in parallel to provide the best coverage to the user.

In addition, Coyote provides support for writing functional *specifications* of the code. These specifications can be written in the form of `monitors`. A `monitor` can only observe the execution of a program but cannot influence it. Syntactically, this means that it can receive messages from any `machine`, but cannot send messages. A `monitor` makes it easy to assert conditions that span multiple `machines`. A `monitor` can also encode *liveness* properties, commonly used in distributed systems to assert for progress [19]. A typical example is, for instance, asserting that a replication protocol *eventually* creates the required number of replicas [8].

To summarize, a system developed using Coyote actors typically implies three activities. First, the system itself must be written using the Coyote actor concurrency model. The Coyote runtime provides APIs to create `machines` and send `events`. Second, external dependencies must be mocked and all sources of non-determinism must be expressed via `RandomBoolean` calls. Third, the user writes tests (exercising the system under a workload) and specifications for asserting correctness.

3 IMPLEMENTING THE POOLMANAGER WITH COYOTE ACTORS

This section outlines the design of the ABS PoolManager and how it is implemented using Coyote actors. The goal is only to provide enough details to impress the complexity of the service, justifying the need to use Coyote, and not to give an exhaustive account of the system. We believe the core reasons behind the complexity are common to many cloud systems.

PoolManager exposes APIs for creating a pool, resizing or deleting an existing pool, as well as canceling a previous resizing operation. We begin by explaining key external services that PoolManager relies on for implementing its functionality before getting into the PoolManager design.

3.1 External Services

Azure Storage. PoolManager operations are naturally long-running because the creation or deletion of VMs takes time (in the order of seconds to a few minutes). If PoolManager fails while creating, say, a pool of size 10 after it has already allocated 3 VMs, then after restarting, it must resume the pending operation and allocate only the remaining 7 VMs. It is important to not lose track of previously allocated VMs (i.e., they must be part of some pool), else ABS risks allocating resources that will never be subsequently used.

Anticipating failures in the middle of an operation, the PoolManager records its progress using Azure Storage [27], a highly available and reliable cloud-scale storage system. Azure Storage offers a key-value storage interface. PoolManager uses REST APIs to read and write information about pools, VMs, jobs, tasks, quota management, etc., as entities (rows) in storage. Azure Storage also provides opportunistic concurrency control using *entity tags* or *ETags*. These are metadata attached to each row (key-value pair). A client can do a conditional write to a row: the row is updated only if the user-provided ETag matches the current value in the row.

Azure Subscriptions. All Azure resources that a customer allocates belong to a subscription, which is a billing entity. Subscriptions, which are managed by the Azure Subscriptions service, can contain accounts for services such as Azure Batch and Azure Storage. Azure imposes limits on how many operations a subscription can perform on resources. Similarly, there are limits on the number of cores one can allocate via VMs in a single subscription. The provided limits are too restricting for running ABS workloads. As ABS is built on public Azure services and needs to allocate resources, it has to own a set of subscriptions (with fairly large limits) and use them to manage resources. By spreading resources across many internal subscriptions, ABS scales beyond what can be achieved with a single subscription.

VMSS. ABS uses VMSS [29] in order to allocate VMs for creating a pool. VMSS offers a service for allocating a collection of VMs, which ABS further wraps into the concept of a pool, for the following reasons:

(1) ABS ties scheduling with resource provisioning. If ABS has a pending request to shrink a pool, and a VM finishes running a task, then ABS proceeds to collect and free the VM. This tight coupling between scheduling and resource provisioning is an important value proposition of ABS.
(2) VMSS imposes VM creation limits per subscription. ABS pools can be much larger than these limits. VMSS also limits the number of operations per subscription. ABS spreads out pool creations across many subscriptions to speed up deployment.
(3) ABS supports pool operations such as stop-resize that are not supported by VMSS. When a user issues a stop-resize operation, ABS moves the corresponding VMSS operations to the background (the customer is not charged) and deletes the extra VMs allocated. The stop operation allows ABS to respond to compute demand more quickly.

The *DeploymentManager* (which is also written using Coyote actors) is a microservice component of ABS that interfaces with VMSS. PoolManager uses the DeploymentManager service to create, grow, shrink and delete individual VMSS collections, also called *deployments* in the rest of the paper.

3.2 PoolManager Design

Design requirements. Central to the PoolManager is the need to manage quotas. Each user has a quota on the maximum number of VMs they can allocate for running their jobs. Further, ABS internally manages multiple Azure subscriptions, each one tied to one region, which limits the number of VMs that can be allocated in that region. Thus, a VM can be allocated for a user only when the user-quota has not been exceeded, and there is some subscription (in some region) whose quota has not been exceeded.

In addition to pool operations such as create and resize, PoolManager can recover VMs that are determined to be unhealthy. The unhealthy signal comes from the ABS scheduler and the PoolManager, in response, deletes the unhealthy VM (by signalling to VMSS) and allocates a new one.

Actors. The PoolManager implementation has over 30 types of Coyote `machines`, totalling over 60K lines of code. Each `machine` is designed to manage the lifetime of a particular resource or to execute a workflow that implements a sub-operation. For example, the `Pool` `machine` manages a single pool, the `PoolServer` `machine` manages a collection of `Pool` `machines`, the `PoolFlow` `machine` allocates resources for a pool, the `Deployment` `machine` manages a single deployment, the `Account` `machine` manages a single account, the `QuotaManager` `machine` manages quota requirements and decides how to allocate across subscriptions.

3.3 PoolManager Operations

CreatePool. To create a pool, PoolManager goes through the following steps: (1) persists the pool properties and puts the pool in resizing state, (2) checks if enough quota is available and tentatively reserves it, (3) allocates resources and creates VMs to match the required pool size, (4) persists deployment and VM information, (5) informs the scheduler about the created VMs so that it can start scheduling job tasks on the VMs, (6) commits the revised leftover quota, (7) updates the pool properties to the final count of resources and puts the pool in *steady state*. An operation is deemed completed once the pool reaches steady state.

Figure 3 shows the workflow that implements this operation to highlight its complexity. Each vertical line corresponds to a Coyote `machine` and arrows represent exchange of `events`. The `AzureStorage` `machine` wraps calls to the Azure Storage service. Arrows to `AzureStorage` represent a read or write of persistent data. `DeploymentManager` wraps VMSS and arrows to this `machine` represent VMSS operations.

In reality, the workflow executes concurrently, responses can arrive in different orders, and timeouts can fire at any time. Due to these reasons, the asynchronous programming model of actors fits naturally with the PoolManager requirements: the `machines` send out requests and field responses as they arrive asynchronously, instead of blocking each time for a response.

ResizePool. A resize-pool operation is similar to creating a pool, except that it may have to grow or shrink existing deployments, in addition to creating new ones. The resize-pool workflow goes through the following steps: (1) persists the new target values and puts the pool in resizing state, (2) checks quotas, if the resize involves growing the pool, (3) for grow operations: allocates resources and creates or grows deployments, then persists the updated information, (4) for shrink operations: works with scheduler instances to identify VMs that can be deleted, (5) commits the revised quota, (6) updates pool properties to reflect final counts and puts the pool in steady state.

CancelResize. The PoolManager allows only one resize or delete operation per pool at a time. To stop this operation, the user can issue a cancellation that goes through the following steps: (1) stopping operations that were creating or resizing deployments, (2) updating the pool size to the previous size plus (minus) any deployments whose creation (deletion) has already committed, and (3) putting the pool in steady state.

After a cancellation is carried out, there may be deployments with extra VMs that have not been freed. These are termed *rogue* VMs. After the pool goes to steady state, a *stabilize deployment* operation starts in the background that

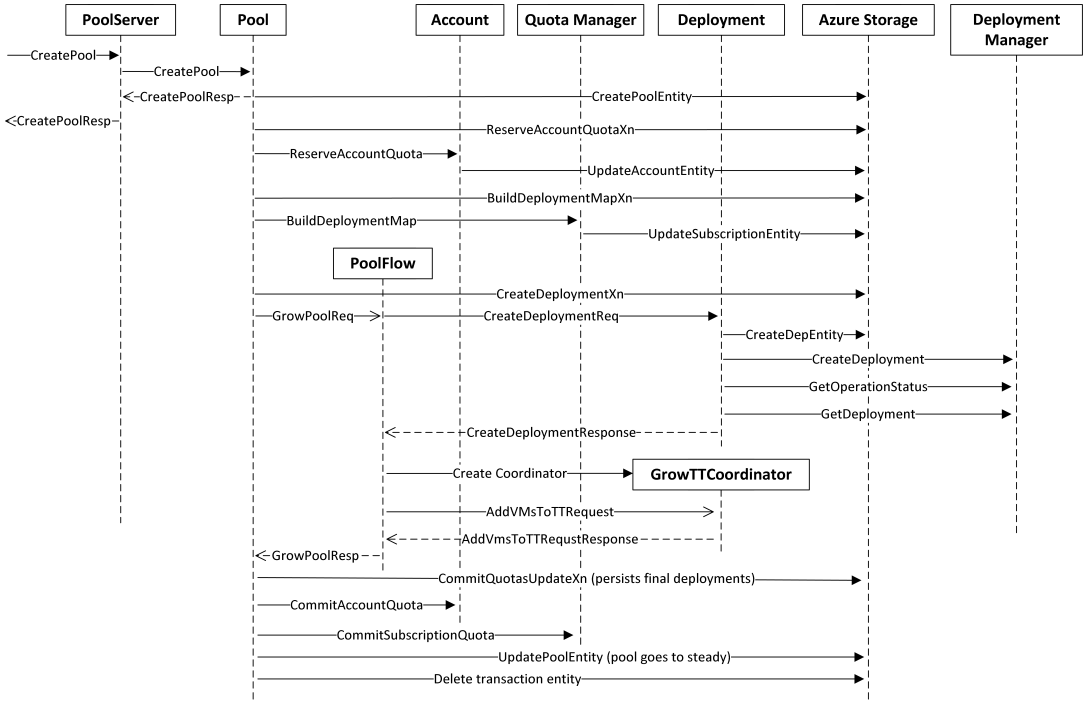

Figure 3: The PoolManager workflow for creating a pool.

asynchronously removes any rogue VMs. This operation first identifies such deployments and puts them in a *stabilizing* state (subsequent resize operations skip deployments that are in this state). It then waits for pending operations on the deployment to complete before issuing fresh operations to remove the rogue VMs. The stabilize-deployment operation also needs to persist progress to storage in order to survive restarts.

An additional requirement is to limit the total number of rogue VMs across all deployments. The `QuotaManager machine` aggregates the rogue-VM count across all deployments. If this number exceeds a threshold, the PoolManager stops new cancellation requests until the rogue-VM count comes down.

4 TESTING THE POOLMANAGER WITH COYOTE

As illustrated in §3, PoolManager involves multiple different operations that can be running concurrently at any point in time. Furthermore, PoolManager has to deal with failures that can happen unexpectedly, and has to correctly resume all pending operations after a failover. Testing such a complex system is where using Coyote makes a difference.

4.1 PoolManager Specifications

The PoolManager specification is around 1,700 LoC, written as a Coyote `monitor` (§2.2) that checks the following properties for each pool.

Liveness properties

(1) If the last client operation was a resize to size N, then the pool *eventually* reaches steady state with size N.
(2) If the last client operation was a delete, then the pool is *eventually* removed and all its allocated VMs are returned back to VMSS.
(3) For a given pool, all pending stabilization, delete and recovery operations must *eventually* complete.

Safety properties

(1) If a pool contains a VM v, then VMSS must have indeed allocated v to the PoolManager.
(2) For every successful create-pool request, a pool entry is created in Azure Storage.
(3) For every successful resize-pool request, the pool target matches what is requested in the corresponding Azure Storage entry.
(4) For every successful delete-pool request, the pool entry is deleted from Azure Storage.
(5) Every deployment enumerated in the Azure Storage pool entry is present in the Azure Storage deployment entry.

(6) Every deployment in the Azure Storage deployment entry belongs to a pool.
(7) Every VM in the Azure Storage VM entry belongs to a pool.

Importantly, the above properties must hold even after a failover. Checking the PoolManager against this specification, especially to get coverage of corner-case behaviors, is a challenging task for several reasons. For instance:

(1) The PoolManager is a concurrent program; its various operations may interleave in many different ways.
(2) The PoolManager interacts with multiple external services and it must be able to handle any valid response from those services. Responses that return error codes (e.g., failure to write to storage) happen rarely, thus are hard to cover during testing. Interactions with external services may time out. The dependence on time further adds complexity.
(3) Failures are non-deterministic events that may happen at any time. Failure-injection tools are often hard to setup.
(4) The specification requires consistency between the PoolManager in-memory state, state stored in Azure Storage and VMSS. Writing such an assertion can be very cumbersome with traditional means because it spans multiple services.
(5) The specification is a liveness property that requires the pool to eventually reach steady state. Executions that get stuck in a loop without making progress are violations of this property and hard to capture using plain assertions.

The Coyote tester was used to continuously test these specifications against production code during its development.

4.2 Mocking External Dependencies

As mentioned in §2, writing Coyote tests requires *mocks* of external dependencies. The ABS engineering team wrote mocks for Azure Storage, VMSS, the ABS scheduler, and a basic system timer (used for encoding timeouts).

Mock Azure Storage. Writing a mock for Azure Storage was easy. It consists of roughly 800 lines of code. The mock has no internal concurrency (i.e., it executes sequentially) but makes non-deterministic choices to expose various error modes. Figure 4 shows a simple illustration of the mock with a `Write` operation. The entire store is modeled as an in-memory dictionary (`store`). The `Write` operation can, for instance, either succeed and write to storage, or it may return one of several error codes. It can return an error code even after writing to the store successfully: a possibility that can indeed happen with the real Azure Storage service. The mock also implements the *ETag* matching logic (§3.1).

```
class MockAzureStorage {
  // Map: StoreName -> PartitionName -> RowKey -> Entity
  Dictionary<string, Dictionary<string,
       Dictionary<string, Entity>>> Store;

  Response Write(WriteContext context, Entity entity) {
    // Does entity size exceed maximum allowed?
    if (IsEntityTooLarge(entity, context.StoreName)) {
      // Raise an error, PoolManager should never do this
      CoyoteRuntime.Assert(false);
    } else if (CoyoteRuntime.RandomBoolean()) {
      return ErrorCode.TIMEOUT;
    } else if (CoyoteRuntime.RandomBoolean()) {
      return ErrorCode.STORAGE_ERROR;
    } else if (!IsEtagMatched(context)) {
      return ErrorCode.ETAG_CHECK_FAILED;
    } else {
      // Perform the write
      Store[context.StoreName] [context.PartitionName]
           [context.Key] = entity;
      if (CoyoteRuntime.RandomBoolean()) {
        // Return failure even when the write is done
        return ErrorCode.STORAGE_ERROR;
      } else {
        return StatusCode.OK;
} } } }
```

Figure 4: A code snippet of the Azure Storage mock.

Mock DeploymentManager. The mock for DeploymentManager, which wraps VMSS, is around 1, 200 LoC. This service has to handle requests for creating, growing and deleting deployments, as well as for deleting specific VMs. The mock uses an in-memory dictionary to track the deployments and the VM instance names. The mock non-deterministically returns HTTP errors including timeouts. Like the real VMSS service, the mock supports multiple levels of failure (e.g., at the operation or HTTP level). One key requirement was to ensure that the mock respects *idempotency* when the real service guaranteed it. For example, once the mock returns success for an operation, then it has to return success if the same operation request is issued again.

Mock scheduler. The mock scheduler is roughly 600 LoC and mimics the ABS scheduler. The job of the scheduler is to schedule tasks onto VMs. From the perspective of the PoolManager, the scheduler has to handle requests for adding or removing VMs and getting VMs that need recovery. The mock scheduler has internal data structures that track pools and VMs (essentially, a `struct` with multiple fields). The mock scheduler can non-deterministically return failures or remove a subset of VMs for a remove-VM request. The mock is required to update the corresponding Azure Storage entries as it accepts or removes VMs. This helps expose possible race conditions in the PoolManager, when a VM that is picked for recovery is also removed during a shrink operation in the PoolManager. (The ETag logic of Azure Storage helps discover and guard against such races.)

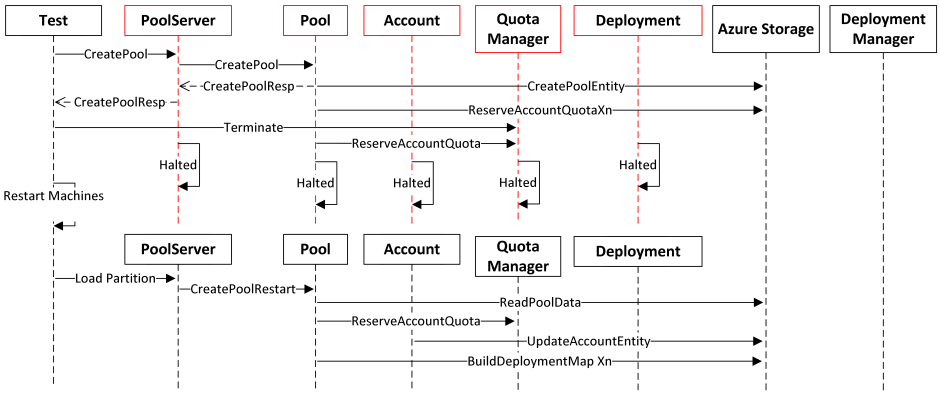

Figure 5: An illustration of failover testing for the PoolManager.

Mock timers. All timers used in the PoolManager were also mocked. ABS engineers wrote an abstract `Timer` class. During testing, `Timer` is implemented using a Coyote non-deterministic choice that can fire the timeout at any point. This helps abstract away time, expressing the fact that correctness of the service does not rely on the particular timeout values chosen. During production runs, an actual system timer is used for implementing `Timer`. Therefore, timeout values can be freely manipulated in the production service in order to optimize performance.

As opposed to testing the PoolManager against production services (e.g., Azure Storage and VMSS), creating mocks is additional engineering cost for using Coyote testing. However, it also has several advantages. First, it clearly lays down the assumptions that the PoolManager is making of these external services. Any deviation from reality can be captured and fixed in the mocks in order to avoid further regressions. Second, all failure modes are made explicit in the mocks using non-deterministic choices. This provides Coyote tester with the hooks necessary to explore rare or exceptional behaviors. Third, the state stored in the external services is captured in the (in-memory) mocks. Thus, asserting for consistency between multiple distributed services is much easier. Furthermore, the use of mocks allows testing to be fast: there is no need to wait on wall-clock time in order to fire timers; no need to go over the network to talk to external services, no need to write to disk to survive (hard) failures, etc.

ABS has various Coyote tests that each exercise different APIs of the PoolManager. All these tests work against the same `monitor` that captures the specification of the system (§4.1). It is then up to the Coyote tester to find a violation of the safety and liveness properties specified by the monitor.

4.3 Failover Testing Technique

The PoolManager failover logic is checked programmatically by mocking failures themselves. The Coyote tests and specifications do not change. The ABS team modeled failures by creating an `event` called `Terminate` (from the perspective of Coyote, this is just a user-defined `event` with no special meaning). Figure 5 provides an illustration of the PoolManager execution when a `Terminate` `event` is injected by the test harness during a `CreatePool` operation. When each `machine` in PoolManager receives a `Terminate` `event` (at an arbitrary point), it forwards the `event` to its *children* `machines` (forwarding of `Terminate` is not shown in the figure for simplicity), waits for them to send a response and once all responses are received, it halts itself. This way, sending a `Terminate` `event` to the top-level `machine` (`PoolServer`) ends up terminating the entire PoolManager. The `Terminate` `event` is only forwarded to the PoolManager `machines` (in red), not the external services (because the test is for the PoolManager failover logic). The failure injection, i.e., the action of sending a `Terminate` `event`, is non-deterministic, thus the Coyote tester will provide coverage by exploring many different possibilities.

After all of the PoolManager `machines` halt, the test harness restarts the service by re-creating the `PoolServer` `machine`. When the `PoolServer` starts, it will read its state from storage, where it will find the state from before the failure (because the mock Azure Storage `machine` survived the "failure"). Thus, the failover logic—*the same logic written to handle real failures in production*—kicks in and the PoolManager resumes the `CreatePool` operation.

For most part, the relationship of a `machine` to its children `machines` is obvious and follows the creation hierarchy: if `machine` A created `machine` B then B is A's child. In some cases, this is more involved, especially when `machines` (legally) halt. In this case, if a `machine`, say A, wishes to halt, then it must first delegate the responsibility of terminating

its children `machines` to some other `machine`. This was done using custom logic that the ABS team designed for the PoolManager. The same termination code is also used for legal teardown of a PoolManager instance, so the team did not see this effort as a test-only overhead.

A key advantage of this technique is that failovers are simply tested at the level of program semantics. It does not require an actual setup with hard failure injection that must crash and re-start the process. The complete engineering of the tests simply becomes a programming activity. It is then much easier for a developer to control and observe failover coverage. For instance, a few lines of code are enough to limit failover testing to one particular region of code. There is no need of resorting to fuzzing or failure-injections tools or a stress-test environment. Debugging is much easier as well because the programmer is given a fully-replayable trace by the Coyote tester, consisting of the actions taken by the system both before and after the failure injection.

5 IMPROVEMENTS IN USING COYOTE

Supporting the development of production services required several engineering enhancements to improve the programming, testing and debugging experience of using Coyote.

Programming improvements. The C# language includes `async` and `await` keywords that make it easy to write asynchronous code [21]. Awaiting on an `async` call packages the current task as a continuation and releases it from the executing thread, so that other tasks can be scheduled. We enhanced Coyote to allow actor event handlers to be `async`: this, in turn, allows handlers to call `async` APIs and `await` on their result without blocking the underlying thread, enabling other actors to be scheduled.

The actor programming model discourages sharing of objects between actors. Message transfers are the only way for actors to synchronize, which can be cumbersome for some tasks, compared to other forms of concurrency. For example, consider the PoolManager task of maintaining the total rogue VM count (§3.3). To maintain this count, the `QuotaManager machine` must communicate with all `Deployment machines` in the system and aggregate their individual counts. Getting a count from each `machine` requires sending an `event` to it, waiting for a response, and defining a handler for the response. This not only increases the programming burden but is not efficient for a simple task such as aggregating counts.

To remedy this situation, we developed a library [36] that allows a Coyote actor to create a *shared object* and freely pass its reference to other actors. These shared objects expose a *linearizable* [16] interface so that multiple actors can issue operations on the (same) shared object without concurrency issues. In production, shared objects are implemented using an efficient lock-free data structure. When running under the Coyote tester, they automatically resort to using actors with message transfers so that the tester does not have to understand any additional form of synchronization. We implemented shared objects for common types such as counters and dictionaries. The design of shared objects showcases the power of mocking: the Coyote actor programming model is not in conflict with low-level or efficient concurrent programming, it simply requires that any concurrency outside of Coyote actors be mocked for testing.

Testing improvements. Even the smallest of concurrent programs can have an astronomically large state space. There are several *search strategies* developed in the research community that target finding common bug patterns fast. Many of these strategies have complementary strengths [10, 39]. The Coyote tester includes multiple search strategies and makes it easy to include new ones. We enhanced the tester to run a portfolio of search strategies in parallel so that engineers, who are likely unaware of these search algorithms, do not have to worry about making the choice.

It is interesting to note that one particular strategy, called PCT [5], worked best for testing the PoolManager. We found that among the various strategies, PCT was able to get the best coverage for failure injection points, which is the place where the `Terminate` (§4.3) event is generated. There are recent additions to search strategies in Coyote that hold further promise in offering even better coverage than PCT [32]. It is an important advantage of Coyote that all users get such advancements for free.

We further enhanced the Coyote tester to parallelize it on the cloud. We used ABS itself: one can create a pool of VMs and run the Coyote tester in parallel on each VM. Coyote testing parallelizes easily: each instance of the tester simply runs a different search strategy (with different parameters and seeds). The typical requirement for many teams was to run the tester for 10,000 iterations, which could be easily achieved on a developer laptop in a manner of minutes. However, occasionally teams chose to run millions of iterations for which cloud-scale testing was important.

Debugging improvements. When the Coyote tester finds a bug, it generates a trace file consisting of all scheduling decisions as well as non-deterministic choices that it made. This trace can be fed back to the tester, in which case, it reproduces the same sequence of choices. To improve the debugging experience, we enhanced the Coyote tester by allowing it to attach a debugger when replaying a trace. Deterministically reproducing reported bugs in a concurrent and non-deterministic system while being able to set breakpoints and step-into the code, was a key value addition that has been appreciated by all developers who have used Coyote so far.

6 EXPERIENCE WITH USING Coyote ACTORS IN PRODUCTION

The positive experience of the ABS team using Coyote invited attention from other teams in Azure. In the two years after the ABS team shared its PoolManager experience with other teams, eight other services built with Coyote actors have gone into production, and several more are in the planning stage. Many of these services share common characteristics with PoolManager: asynchronously arriving requests that must be processed concurrently in a non-blocking fashion, multiple distributed data sources that must be kept consistent with each other, and interaction with several other services.

Each team echoed two *key advantages* of using Coyote actors: (1) the programming model allowed them to implement a service at a higher-level of abstraction, resulting in a cleaner design and code that is easier to maintain, extend and explain to new team members; and (2) the *high-coverage testing* allowed them to exercise many corner cases and find several high-severity bugs before deployment. The rest of this section summarizes the experience of all these teams from using Coyote actors to design, implement and test their cloud services.

Benefits in design and implementation. Several teams reported that using the Coyote actor programming model helped them implement services that closely match their initial high-level (whiteboard) design. A senior Azure engineer said: *"the design maps very closely to the actual code, usually I see a much bigger delta between design and finished product"*. Closing the gap between design and implementation, allowed teams to easily create diagrams such as in Figure 3 that provide not only a detailed understanding of the workflow implementing each operation, but also the expected communication between Coyote actors. These diagrams were useful in explaining the design to other team members. Other benefits of using the actor-based approach of Coyote include:

- The `events` sent between actors have to be clearly defined in the implementation. Further, no data can be shared between `machines` unless explicitly sent via an `event`. Both of these helped improve code abstraction and readability.
- Coyote actor code is naturally non-blocking (asynchronous), so there is no need for explicitly locking resources or managing a thread pool. These benefits together were a welcomed relief over typical multi-threaded code with threads and locks.
- Actors are lightweight and event-based, which can lead to significant performance gains. One of the teams reported that their previous design relied on polling, which consumed too many CPU cycles. With actors, they were able to write fully reactive code that allowed them to scale to much larger workloads. For instance, they were able to hold up to 200,000 Coyote actors before seeing the CPU reach 80% utilization.

It is worth noting that Coyote actors was not perceived to be an "arcane" technology only used and understood by the most senior engineers. A junior engineer reported: *"being a new developer to the team, one of the first few things I worked on was Coyote actors, it was really quick to onboard and writing actual code is simple and straightforward"*.

The importance of mocking. Cloud services typically operate by communicating with their environment, which can consist of other services, as well as resources such as network and storage. To simplify the development process, teams would initially create *interfaces* for all external dependencies of their service, and then provide simple mock implementations of these interfaces (e.g., the ABS team created a mock for Azure Storage as seen in Figure 4). Importantly, Coyote mocks allow developers to express nondeterministic behavior that can be controlled during testing. A large part of the development of each service was done against these mocks in a test environment.

The efficacy of Coyote testing relies on how closely mocks model the real behavior. Any deviation can lead to missed bugs (when mocks do not exercise some possible behavior) or even false alarms (when mocks exhibit some behavior that is not possible in reality). Interestingly, each time there was an issue in production that was not found by the Coyote tester, it turned out to be either due to a missing test (some workload was not exercised by the Coyote tests) or an incomplete mock. It was *never* the case that the Coyote tester could have found the bug, but missed it because of lack of coverage. In these cases, developers would add more tests or patch the mocks. The teams knew about these tradeoffs before deciding to use Coyote. Maintaining the mocks was an iterative process and deviations were fixed over time as they got noticed. The initial mocks simply followed the available online documentation and gradually got more detailed over time, and thus more effective. This *pay-as-you-go* model of writing mocks was important to avoid front-loading the implementation with mocking effort.

Some of the services that were mocked included: Azure ServiceBus [26], Azure Cosmos DB [24], Azure Storage [27], various networking services [30] and resource providers [25]. There was sharing of mocks between teams, but each team ended up owning its own mocks so they could customize them in ways most relevant for their service.

To illustrate one example, the Azure blockchain team built a service using Coyote actors that is designed to hide the complexity of blockchains from users [23]. It deals with

issues of submitting transactions exactly once, hiding forks and rollbacks from users, etc. In order to test this service, the blockchain team wrote a mock of a blockchain network itself that nondeterministically created forks and rollbacks. The ability of authoring Coyote tests for exercising such scenarios was very valuable.

The value of systematic testing. Teams typically focused on writing end-to-end Coyote tests for their services instead of writing standalone tests for individual Coyote actors. This was enabled by the fact that systematic testing can deal with concurrency and all declared sources of non-determinism, after which the Coyote tester was responsible for getting coverage. This freed the developer from breaking down an end-to-end scenario into smaller units and assertions for testing individual actors.

Furthermore, it was easier to write a specification for end-to-end scenarios. A common specification among many services was to validate liveness properties: that it would *eventually* complete the client request, even in the presence of failures. Coyote enabled writing such end-to-end specifications just once, and reuse them for all relevant tests. The tests themselves only vary in the client workload they execute. Further, Coyote tests would typically run much faster than stress testing or complex simulations. For example, exercising failover in PoolManager (§4.3) takes approximately two minutes for 10,000 iterations in an Intel Core i7 laptop with 4 cores and 16GB RAM.

Developers frequently executed Coyote tests to validate safety and liveness specifications as they made code changes, ensuring that the implementation never regressed. Some teams reported that the Coyote tester helped them find several high-severity bugs before deployment that would have been hard to find using conventional means, and would have resulted in loss of business if they occurred in production. In the words of a service architect, using Coyote they *"found several issues early in the dev process, this sort of issues that would usually bleed through into production and become very expensive to fix later"*.

Once the service code was reasonably functional in a test environment, a team would re-implemented the mocked interfaces (as discussed above) to communicate with the actual external components. Then the code—*the same code that was systematically tested*—was deployed by using this implementation of the interfaces. Most teams reported that once a feature was tested with Coyote, it would *just work* when put into production.

Scope for further improvements. We contributed several improvements to Coyote to help adoption (§5), but other challenges remain that were brought out by engineering teams. We list some of them here. One challenge was that the cost of writing mocks can be unknown upfront. While mocking is a standard practice in software engineering, writing functional mocks of complex services can be difficult. For instance, one team owned a service that relied heavily on transactional aspects of a backend database. Mocking this interface, while preserving the transactional guarantees, was too difficult. The team eventually used an out-of-process simulation of the database in order to do (standard) testing. While it is possible to integrate this simulator with Coyote tests, the result would be cumbersome and it would considerably slow down systematic testing.

Another challenge is that testing is limited by the quality of the mocks. A constant worry by teams was that if their mocks were incorrect (or incomplete), then Coyote tests would not exercise behaviors that could happen in practice. In this case, traditional integration testing would still be required, adding to the number of tests that the team has to maintain. A usual compromise was to do as much testing with Coyote as possible, but still maintain some integration tests.

One source of frustration using the Coyote tester was when it did not cover a path that the developer expected. The tester does not provide any evidence why some behavior did not happen and the developer would have to manually debug and find out if the mistake was in their understanding, or in lack of coverage by the tester.

Finally, using the Coyote tester requires supplying a limit on the number of test iterations. Knowing what number is a good bound can be difficult. The ABS team would occasionally run a very large number of iterations, but they found that the tester almost never reported a bug after 10,000 iterations. So they finally settled on this bound for their inner-loop testing.

7 RELATED WORK

Systematic testing tools. The research community has been long interested in finding bugs in distributed systems. Previous work showcased how *systematic testing* (ST) tools and search techniques can successfully find deep concurrency bugs [20, 33, 34, 42, 44]. Many of these search techniques have been adapted almost directly by Coyote [8]. However, it is critical to consider the nature in which these techniques are exposed to a user.

In order to reduce user effort, ST tools have mostly targeted existing systems without modification. This approach requires ST tools to take over *all* sources of non-determinism in these systems. Obtaining such level of control is difficult because the API surface of such systems can be very broad. For example, Chess [33] targeted the testing of concurrent multi-threaded programs on Windows. To control thread interleaving, Chess had to interpose at the Win32 API level (via

stubs) and reliably identifying all sources of thread synchronization. It was necessary to get these stubs right, without which the tool would be flaky or even deadlock, leading to user frustration. The effort required to maintain these stubs was too large, and Chess went out of support without seeing user adoption, even though it found numerous bugs. Instead of targeting unmodified systems, Coyote spells out how a new system must be built from the outset. Coyote actor concurrency is simple to control: its only about actor creation and message passing. Any use of external nondeterminism must be mocked: an exercise that is much easier for a programmer who controls the design of their code.

Systems-level imposition can also be slow. For example, SAMC [20] takes roughly 6 hours for doing 1,000 test iterations of Cassandra [13] because it must bring up the actual database in each iteration and inject failures via actual system crashes. The use of mocks to model real-world interactions and failures offers speed: roughly two minutes for 10,000 iterations of PoolManager with Coyote tester (§6).

Modeling languages. Researchers have argued for principled design of distributed systems through the use of modeling languages such as TLA+ [19] or Promela/SPIN [18]. TLA+ has been extensively used to model and specify distributed protocols and algorithms [35]. One can apply inductive reasoning to these model (which is harder) or use push-button model checkers (which is easy). Modeling languages are useful for validating the high-level design of a system. However, they do not help with the actual implementation. As new features are added, the implementation often diverges from the initial design that was modeled. Coyote bridges the gap between design and implementation: what you test is what you execute.

Formally verified systems. Recent research efforts have focused on developing formally-verified systems [14, 43]. Building such systems involves using a high-level language that can generate executable code, as well as contain logical assertions that mark *inductive* system invariants. The inductiveness checks, as well as the check that the invariants imply the system specification, are all discharged by a theorem prover to establish the proof of correctness. Examples of such systems include: crash-tolerant file systems [6], simple operating systems [15], distributed key-value stores [14] and protocols [4, 43]. Although this line of research is exciting, all of these systems have been developed in academic settings. Inductive reasoning requires deep understanding of formal logic and that is outside the scope of education that most software developers receive. This constitutes the biggest bottleneck for adoption of these practices in the industry.

Coyote removes the need for theorem proving. Developers must write specifications of their program to find bugs, but there is no need for inductive reasoning. At the most, one must learn the concept of liveness properties. The emphasis of Coyote in programmability allows engineering teams to use Coyote effectively without having a researcher in the loop. Coyote guarantees are not as strong as full verification: Coyote testing is still an argument of coverage. However, as this paper shows, Coyote testing has (so far) not missed a bug due to lack of coverage.

Previous work on P*, P#, and Coyote.* Coyote is an evolution of the P# project. P# was designed specifically for the *communicating state-machine* programming model, which was itself inspired from the P language. Coyote now supports multiple programming models.

Previous work on P# focused on defining the framework [7] and showcasing its bug-finding abilities [8, 31], but only targeted existing systems. In our experience, convincing teams with just bug-finding capabilities alone was not enough. It was much easier to convince them once we had a success story (with Azure Batch) that demonstrated *overall faster development time*, along with increased service quality.

The P language [9] consists of its own language definition and compiler, and it is a different design point compared to P# or Coyote, each of which are C#-specific solutions. P has been used in the past for Windows Device Driver development [9] and is currently being used in Amazon for creating formal models of distributed protocols involved in AWS's S3 system [3].

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Amazon. 2012. Summary of the AWS service event in the US East Region. http://aws.amazon.com/message/67457/.

[3] Amazon. 2021. Use of Automated Reasoning for S3 Strong Consistency. https://www.twitch.tv/videos/962963706?t=0h26m57s. [Online; accessed 14-September-2021].

[4] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *SNAPL*. 1:1–1:12.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*. 167–178.

[6] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-performance Crash-safe File System Using a Tree Specification. In *SOSP*. 270–286.

[7] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. Asynchronous programming, analysis and testing with state machines. In *PLDI*. 154–164.

[8] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. 2016. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *FAST*. 249–262.

[9] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *PLDI*. 321–332.

[10] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic testing of asynchronous reactive systems. In *FSE*. 73–83.

[11] Forbes. 2018. 83% of enterprise workloads will be in the Cloud by 2020. https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020.

[12] Forbes. 2019. Public Cloud soaring to $331B by 2022 according to Gartner. https://www.forbes.com/sites/louiscolumbus/2019/04/07/public-cloud-soaring-to-331b-by-2022-according-to-gartner.

[13] Apache Foundation. 2019. Cassandra. http://cassandra.apache.org/.

[14] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving practical distributed systems correct. In *SOSP*. ACM.

[15] Chris Hawblitzel, Jon Howell, Jay Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI*.

[16] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI* (Boston, MA). 295–308.

[18] Gerard Holzmann. 2011. *The SPIN Model Checker: Primer and Reference Manual* (1st ed.). Addison-Wesley Professional.

[19] Leslie Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16, 3 (1994), 872–923.

[20] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI*. 399–414.

[21] Microsoft. 2019. Asynchronous programming in C#. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/.

[22] Microsoft. 2019. Azure Batch: Cloud-scale job scheduling and compute management. https://azure.microsoft.com/en-in/services/batch/.

[23] Microsoft. 2019. Azure Blockchain Service. https://docs.microsoft.com/en-us/azure/blockchain/service/overview.

[24] Microsoft. 2019. Azure Cosmos DB. https://azure.microsoft.com/en-in/services/cosmos-db/.

[25] Microsoft. 2019. Azure Resource Manager. https://docs.microsoft.com/en-us/azure/azure-resource-manager/.

[26] Microsoft. 2019. Azure Service Bus. https://docs.microsoft.com/en-us/azure/service-bus-messaging/.

[27] Microsoft. 2019. Azure Storage. https://azure.microsoft.com/en-us/services/storage/.

[28] Microsoft. 2019. Azure Subscriptions. https://docs.microsoft.com/en-us/azure/azure-subscription-service-limits.

[29] Microsoft. 2019. Azure Virtual Machine Scale Sets. https://azure.microsoft.com/en-in/services/virtual-machine-scale-sets/.

[30] Microsoft. 2019. Azure Virtual Network. https://docs.microsoft.com/en-us/azure/virtual-network/.

[31] Rashmi Mudduluru, Pantazis Deligiannis, Ankush Desai, Akash Lal, and Shaz Qadeer. 2017. Lasso detection using partial-state caching. In *FMCAD*. 84–91.

[32] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-based controlled concurrency testing. *Proc. ACM Programming Languages* 4, OOPSLA (2020), 230:1–230:31.

[33] Madanlal Musuvathi and Shaz Qadeer. 2008. Fair Stateless Model Checking. In *PLDI*. ACM, 362–371.

[34] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*. USENIX, 267–280.

[35] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73.

[36] Microsoft Research. [n.d.]. Coyote Actor SharedObjects Documentation. https://microsoft.github.io/coyote/#concepts/actors/sharing-objects/. [Online; accessed 12-September-2021].

[37] Microsoft Research. 2020. Coyote: Fearless coding for reliable asynchronous software. https://github.com/microsoft/coyote.

[38] Gregory Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3* (2002).

[39] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2016. Concurrency Testing Using Controlled Schedulers: An Empirical Study. *TOPC* 2, 4 (2016), 23:1–23:37.

[40] Ben Treynor. 2014. GoogleBlog – Today's outage for several Google services. http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html.

[41] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *SOCC*. 5:1–5:16.

[42] Jiří Šimša, Randy Bryant, and Garth Gibson. 2011. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *SPIN*. Springer-Verlag, 188–193.

[43] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*. ACM, 357–368.

[44] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*. 213–228.