Towards Ensuring Cloud Reliability Through Different Objectives

Zhiqing Zhong
223040261

***Abstract*—Cloud reliability is essential today. Although ensuring cloud reliability can be challenging and sometimes labor-intensive, unexpected issues can lead to significant financial loss. This report presents three works that enhance cloud reliability. One project tests cloud system operators to reduce potential failures. Another allows servers with failed components to continue hosting virtual machines (VMs) without repair, effectively masking degraded capacity and performance. The third project enables timely diagnosis of application-network anomalies facilitating quick failure recovery. All three projects have been published in top venues in operating systems.**

***Index Terms*—Cloud Reliability, Failure Recovery, Incident Diagnosis**

I. Introduction

Cloud systems are growing in scale and demand beyond what human-based operation can reliably, continuously, and efficiently manage. Modern cloud systems are increasingly being managed by operation programs, termed operators [1], that automate labor-intensive operations. Operators of cloud management platforms like Kubernetes [2], Twine [3], and ECS [4] implement declarative interfaces based on state reconciliation. An operation declares the desired system state and the operator automatically reconciles the system from its current state to the declared state. This "cloud-native" operator pattern simplifies operations and improves efficiency.

The rapid development and deployment of operators make their quality assurance a pressing need—operation correctness is critical to system reliability [5]. A buggy operator can impair correctly implemented systems in production. Compared with human operator mistakes—major causes of system failures [6], [7]—bugs in operators have more magnified impacts due to the nature of automation and widespread software reuse. In fact, buggy operators caused many recent production incidents [1], [2], [8].

Server hardware failures are quite frequent in cloud platforms. For example, a typical cloud server relies on at least 24 DIMMs, six SSDs, six fans, and two CPU sockets [9]. Even assuming optimistic annual failure rates of 0.1% per DIMM and 0.2% per SSD, 22% of servers will have at least one failure during the typical 6-year lifetime of a cluster. In practice, repair rates are typically even higher.

Cloud regions host the core business systems of Alibaba and serve billions of customers worldwide [10], [11]. The key challenge of operating Alibaba cloud is timely detection and diagnosis of application-network anomalies, to ensure service level agreements (SLAs) and avoid customer, reputation, and revenue losses caused by SLA violations [12].

This report presents three techniques to address network incidents, hardware failures, and bugs in cloud operators. To tackle the network issues, we introduce the experience in designing and deploying the Application-Network Diagnosing (AND) system in Alibaba cloud. AND exploits TCP retransmissions (rest xs) to extract anomalies with low overheads, capture problems at a (micro)service level, and correlate applications with platform/infrastructure layers end to-end. Existing works [6] also collect TCP retxs and statistics at end hosts for anomaly detection and diagnosis. We observe several deployment challenges that hinder the direct usage of these systems in a cloud-native environment. To tackle hardware failure, we introduce Hyrax, the first implementation of the fail-in-place paradigm for cloud computing servers. In a multi-year study of component failures across five server generations, we find that sufficient redundancy in existing servers can overcome the most common memory and SSD device failures. While existing diagnostics can only identify a subset of component types, we empirically find that they are 95% accurate. We identify hooks in deployed firmware that enable deactivating components in ways that overcome many failure possibilities (e.g., dirty or corroded connectors or chip failures). Finally, Hyrax adds a degraded server state and corresponding scheduling rules to a production control plane to support servers with deactivated components. To detect operator bugs, we introduce Acto, the first automatic technique and tool for end-to-end testing of cloud system operators. Acto automatically generates end-to-end tests to check three operation correctness requirements: the operator (1) always reconciles the managed system to desired states, (2) performs managed system recovery from undesired or error states by rolling back to a previous good state, and (3) should be resilient to misoperations (i.e., operation errors) by preventing them from driving the system into error states.

AND has been deployed in Alibaba cloud for over three years. It processes more than one billion retxs events per hour and reduces data volume by orders of magnitudes. The routing model achieves 95% recall/accuracy in detecting/routing anomalies to target network teams in the long-term evaluation. AND enables minute-level anomaly detecting and routing. AND also achieves high coverage of anomalies in the end-to-end path, from networks to end hosts and even abnormal application behaviors, and complements the fade area of existing systems.

Hyrax has been deployed for a few months on a subset of Azure clusters and a small set of component types. We report

on its effectiveness on real failures and use microbenchmarks and large-scale trace-driven simulations to extrapolate a full deployment over six years. Our experience demonstrates that the fail-in-place paradigm is practical under real-world platform constraints.

We evaluated Acto on eleven popular Kubernetes operators of various kinds. Acto found 56 new operator bugs in total, among which 42 have been confirmed and 30 have been fixed. Acto also found six bugs in Kubernetes and in the Go runtime that affected multiple operators (all have been confirmed or fixed). The detected bugs lead to severe safety and liveness issues, affecting not only the operators but also the reliability and security of the managed systems. Lastly, Acto finds many vulnerabilities to misoperations. Acto tests all these operators within eight hours (a nightly run) on a cluster of eight machines; five of eleven operators only need one machine. Acto has few false positives: Acto reports no false alarm and Acto has a 0.19% false alarm rate.

II. Background

A. Modern Cloud Operation Programs

Operation programs (i.e., operators) for modern cloud management platforms like Kubernetes [13], Twine [14], and ECS [15] follow a declarative, state-reconciliation design pattern. An operation declares a desired system state and the operator automatically reconciles the system to the declared state. This design pattern simplifies system management operations by removing the need to write ad hoc, imperative scripts for different one-off tasks. The pattern also makes system management declarative and intent-driven. We give a brief overview of the pattern, using Kubernetes [16] as a representative example.

1) Declarative operation interface.: In Kubernetes, operators expose a declarative interface in the form of custom resources (CRs) [17]. A CR defines a system resource and its properties that can be modified to manage that resource. A state declaration specifies property values in a CR. Figure 3 shows an example of desired-state declarations for ZooKeeper; it specifies primitive properties like replicas and image, and composite properties like persistence which has sub-properties. A ZooKeeper operator reconciles a managed ZooKeeper cluster to satisfy the declared state. Management operations are expressed by changing one or more property values in a CR. Kubernetes operators maintain CR definitions in the OpenAPISchema format [18], which defines constraints on each CR property (e.g., data type and data range). Operations that change a CR are first validated against the specification by the API servers, before being forwarded to the operator.

Kubernetes operators maintain CR definitions in the OpenAPISchema format [19], which defines constraints on each CR property (e.g., data type and data range). Operations that change a CR are first validated against the specification by the API servers, before being forwarded to the operator.

2) Operator design pattern: Kubernetes operators follow the state-reconciliation pattern of modern cloud management platforms and control planes, such as Kubernetes, Borg, Omega, Twine, and ECS [1], [4], [20], [21]. An operator continuously reconciles the managed system from its current state to a newly declared desired state, if the current state does not match the declared state. The management platforms maintain their current system states in a collection of state objects in strongly consistent datastores (e.g., etcd [22]). Every entity in the system, such as a pod, a volume, and a stateful set (representing a stateful system), has a corresponding state object. State objects have uniform APIs and consistent data schema, making them highly interpretable and extensible [23].

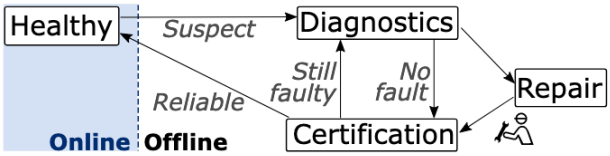
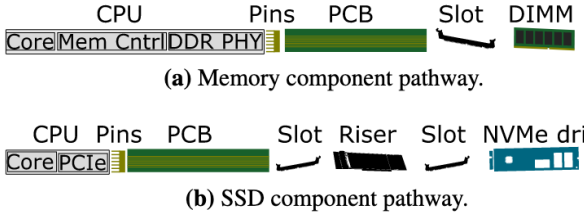


Fig. 1. At Azure, servers are either online and serving VMs, or offline and being repaired. Repairs take between 3 and 190 days at the 50-th and 99-th percentile, respectively.



(a) Memory component pathway.

(b) SSD component pathway.

Fig. 2. A component can appear faulty due to other component faults along the path between a core and the actual component. We call this a pathway that typically spans the socket and pins, printed circuit board (PCB), and slots/risers to the actual component like the NVMe SSD or memory DIMM.

B. Cloud hardware failure

1) Repair workflow: A software agent called Server Health Monitor (SHM) checks server error logs and component types, counts, and capacity for deviations from the expected (homogeneous) configuration. If the SHM suspects any kind of fault, the server is marked as "offline", which signals the VM scheduler to filter out this server (Figure 1). VMs are migrated away or gracefully evicted. The server is then rebooted into a diagnostics environment. If diagnostics finds a hardware problem, it immediately creates a repair ticket [24]–[26].

Repair tickets can point to a specific component pathway (like DIMM 4, Figure 2a) or require a manual diagnosis. After a technician resolves a ticket, e.g., by reseating connectors or swapping out components, the server is tested again to certify reliability (certification step). A reliable server is marked "online" and again becomes a candidate for hosting VMs.

2) Impact of all-or-nothing repairs on TCO: Server repairs are a significant component of total cost of ownership (TCO). The main components of TCO are CapEx (capital expenditure for the purchase of servers, networking, cooling, and power infrastructure) and operational costs due to energy and power (estimated at 6% of CapEx per year [6], [27], [28]), and maintenance (estimated at 5% of CapEx per year for each

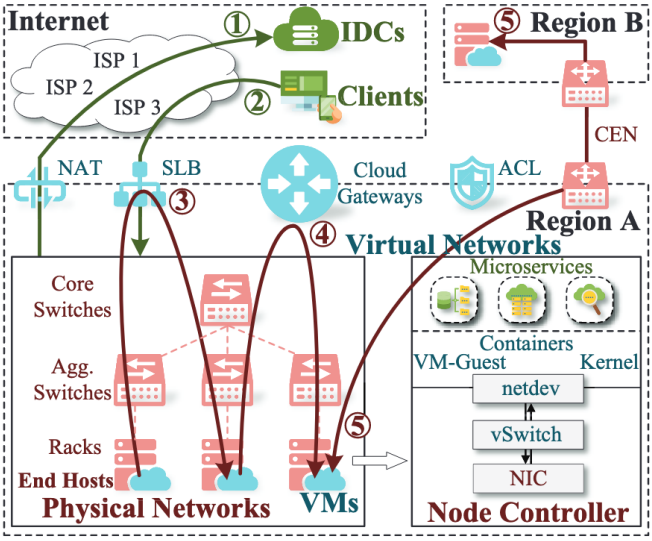


Fig. 3. Overview of cloud traffic.

server [19], [29]). Maintenance costs are largely made up by technician salaries and cover maintenance of all datacenter components. At Azure, server repairs account for about half of technician work hours in the all-or-nothing operating model. Server repairs thus account for 9% and 12% of total cost (TCO) for server lifetimes of 6 and 10 years, respectively.

Repairs are also known to be slow [30]. At Azure, 2% of servers are waiting for repairs at any given time in the all-or-nothing operating model.

3) Server hardware: Figure 4 shows a typical cloud server configuration [48]. Variants of this base architecture include one or two NICs and 24-32 DIMMs; most servers use a single NIC. We note that the component count for some component types is larger than one. We refer to these as degradable components as they do not represent a single point of failure.

We note that hardware components internally contain redundancy, such as spare blocks in SSDs [5], [31]–[34]. Moreover, the operating system and hypervisor at Azure employ an aggressive policy for offlining memory pages to mask faulty cachelines. A repair ticket is generated for a component only when the above mechanisms cannot resolve the problem.

C. Network Operations

1) Overview of cloud traffic: Cloud applications go through each layer of components in the end-to-end path. As shown in Figure 3, applications run on containers and VMs, and communicate via kernel stack and virtualized netdev [35]. The traffic is then forwarded via vSwitch [36] and physical NIC at host machines or node controllers (NCs). The cloud gateways [5] as the cores of virtual networks perform stateless and stateful network functions, including forwarding gateways among virtual private clouds (VPCs), stateful load balancing (SLB), network address translate (NAT), access control list (ACL). The physical network consists of multilevel switches/links, and the cloud enterprise network (CEN, a dedicated leased line network between cloud regions).

On this basis, we summarize five common scenarios of cloud traffic (Figure 3): (1) Cloud services actively access internet data centers (IDCs). (2) User clients request services hosted by the cloud, which return responses. (3) Cloud services access other intranet services via SLB. (4) Cloud services access each other directly via forwarding gateways. (5) Cloud services access each other cross regions.

2) Operational experience: How to estimate the impact of anomalies on real application traffic? The networking teams build large-scale probing systems to monitor virtual/physical networks [29], [37]. However, probing systems cannot tell whether application traffic is affected by anomalies and even overlook severe failures. Specifically, they probe each node/link with equal weight, while partial nodes/links carrying critical (micro)services that many applications depend on should have larger weights. Probing results on these critical nodes/links may be covered by noises of network jitters. We observe many cases where failures in critical (micro)services are not perceived by probing systems. Real case studies of SLB and Redis services are presented. Other monitors on the platform or infrastructure layer face similar problems. Even though anomalies are detected, they cannot tell whether or how many applications are affected.

III. Methodology

A. AND

we introduce the design and deployment of AND in practice to resolve the above challenges. Figure 8 shows the overall architecture of AND incorporating anomaly collecting, detection and diagnostic. The process from anomaly collecting to routing takes less than 1 minute.

1) Challenge 1: Anomaly Collecting: AND monitors retxs at end hosts via a dedicated tool, namely nBPF. nBPF designs eBPF-based kernel filters to extract prominent anomalies that impact application performance, i.e., fails of connection establishment and retransmission timeouts. nBPF also devises user-space filters to record accurate retxs counters for anomaly detection and sample retxs details (TCP 5-tuple and even process info) for anomaly diagnostic. Last but not least, nBPF proposes statistics of retxs events per (micro)service (SRC) and per traffic scenarios (DST) to delimit the scope of anomalies in the collecting phase. nBPF effectively filters out $\geq$ 90% of retxs and achieves low overheads in extreme stress tests with million connections. The filtering rules also ensure a high coverage rate of anomalies and help anomaly routing among multiple scenarios.

2) Challenge 2: Anomaly Detection: AND then performs real-time anomaly detection on time series of retxs counters. AND adopts multi-level time-series clustering to distinguish distinct retxs patterns with respect to frequency, stability, seasonality, etc. According to the clustering results, AND designs lightweight feature engineering and extracts normalized features for anomaly detection. The compute-intensive clustering and model training are conducted in the offline

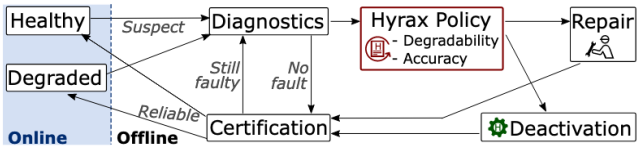


Fig. 4. Server states in Hyrax.

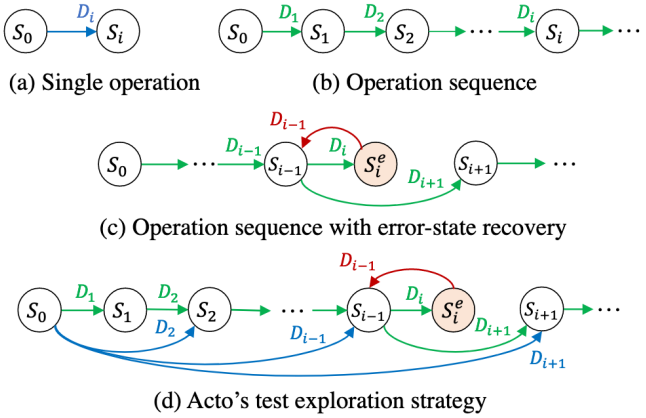


Fig. 5. State transitions of different test strategies.

phase, while the feature extraction and anomaly detection are executed in real-time. Finally, AND achieves minute-level detection for millions of data streams (time series of millions of IPs $\times$ multiple scenarios). AND extracts $\leq 1\%$ abnormal IPs from 'noisy' time series of retxs and guarantees high recall in anomaly detection.

3) Challenge 3: Anomaly Diagnostic: The abnormal IPs are aggregated by application/network attributes and exported to the diagnostic process. The key insight is that anomalies in multiple scenarios, e.g., intermediate networks, end hosts, etc., exhibit different distributions of retxs, after correlating retxs details with application/network attributes. AND builds a supervised anomaly-routing model using single retxs metrics and designs feature sets of anomalies for multiple target teams or scenarios. The routing model also expands the training sets and executes re-trains in a self-iterative way. In the long-term evaluation in the production cloud, AND achieves $\sim$ 95% recall/accuracy in detecting/routing network anomalies. The routing model helps to locate anomalies in specific scenarios and embodies generalization ability to more problem domains.

B. Hyrax

Hyrax is a concrete implementation of the FIP idea and the first FIP system at a cloud provider. Hyrax implements a new "degraded" online server state on servers and in the control plane and changes multiple aspects of the offline workflow at Azure. Currently, Hyrax supports three degradable component types: memory, SSDs, and fans.

Figure 4 provides an overview of server states in Hyrax. After a server is marked as suspect, results from Diagnostics are used by the Hyrax Policy to decide whether to degrade or repair. This policy applies first filters for degradable component types. Second, it verifies that diagnostics points to a specific pathway within this component type. Third, it applies a threshold on how many components of each type can be degraded. Degraded servers are created by deactivating the faulty component pathway. Repairs are scheduled for undegradable component types, when diagnostics cannot identify the faulty component pathway, or if deactivation would cross the policy's threshold. Degraded and repaired servers are subject to extensive testing before becoming available for hosting VMs (online).

Hyrax achieves $C_{Efficiency}$ via the policy's thresholds. Currently, we never deactivate more than two components of any type. Empirically, we find that this is sufficient to prevent resource stranding.

Hyrax achieves $C_{Performance}$ by characterizing how deactivating components affects VM performance for different VM types. This allows Hyrax to decide whether the remaining healthy components are sufficient for the server to continue serving VMs and which VM types it can serve without impacting user experience. Hyrax modifies the VM scheduler such that only the VM types whose performance requirements can be met are scheduled on the degraded server.

Hyrax minimizes repair tickets because many servers that are degraded instead of repaired will not encounter another fault during their deployed period. If degraded servers encounter another fault that cannot be degraded, Hyrax issues a single repair ticket and technicians repair all faults on the server at once. We call this technique "mini-batching". Minibatching effectively amortizes technician work like the journey to the server's rack, identifying and opening the server, manual diagnosis, and record keeping.

Hyrax achieves $C_{Capacity}$ in two ways. First, the capacity an individual degraded server can lose is limited via the policy's thresholds. Second, undegradable servers are not permanently left offline without repairs.

C. Acto

1) Realizing State Transitions: During a test campaign, Acto automatically generates a new state declaration D_{i+1} based on the current system state S_i to realize a state transition, S_i, $D_{i+1} \rightarrow S_{i+1}$. Test campaigns start from the initial state S_0. Acto triggers state transitions with the goals to: (1) cover all properties exposed by the operation interface, and (2) exercise representative operation scenarios based on property semantics.

Acto systematically exercises all the properties that are defined in the operation interface. Each new D_{i+1} changes one property in the current state S_i and any other properties that are needed to satisfy predicates on property relationships. Specifically, Acto selects a previously untested property and uses it to declare a new desired state. The end state after one transition, becomes the start state for the next transition

(Figure 5b). All state declarations collectively change every property at least once during a test campaign.

Acto tests different scenarios based on the semantics of the changed properties. (Acto automatically infers these semantics). For example, Acto tests the scale-up-and-scale-down and the scale-downand-scale-up sequences if a property represents the number of replicas. Acto also tests different pod assignments that trigger the operator to re-configure or re-deploy managed systems differently. This scenario-driven approach allows Acto to focus on a small number of representative states, instead of the very large set of all possible property values. We implement scenarios as plugins that can be extended or customized; users of Acto can add more plugins.

In addition to valid operation scenarios, Acto also generates misoperations, each of which triggers a state transition to an error state, S^e. For example, Acto generates misoperations that (1) scale the replicas beyond the total number of available physical resources, and (2) set unsatisfiable affinity rules. Acto uses misoperations to check if an operator (1) is resilient to operation errors, and (2) can recover from undesired or error states. Acto's oracles check the former (is the system in a state S^e?). Acto checks the latter by rolling back S^e to the most recent healthy state. Misoperations that declare semantically erroneous states could escape constraint validation. A correct operator should not carry out an erroneous operation or at least should be able to recover from operation failures.

2) Extracting Constraints: Extracting Constraints. The operation interface specification defines syntactic validity constraints on state declarations. For example, Kubernetes' OpenAPISchema specification defines constraints on all supported properties. Acto uses these constraints to ensure that all property values in declared desired states are syntactically valid. (Invalid declarations would likely be directly rejected by the API servers before reaching the operator.) For composite properties, Acto uses composite constraints like required properties and also derives constraints from the sub-properties. For primitive properties, Acto uses constraints like the type, min/max values (for numeric types), length (for string type), regular expression patterns, etc.

3) Inferring Property Semantics.: To exercise different scenarios, Acto changes properties based on their semantics. Acto infers the semantics of a property in the interface specification by mapping it to a set of resource types in the Kubernetes core APIs. Such mapping is feasible because many operations for property changes are eventually delegated to Kubernetes core services.

4) Inferring semantics from property structure: Acto exploits the insight that property structure is effective for mapping to properties in the Kubernetes core resource specification. Specifically, all Kubernetes core resource types have unique structures. Figure 5 exemplifies how Acto infers semantics from the property structure: CassOp has a cassandraDataVolumeClaimSpec property with the same structure as the VolumeClaimTemplates property in Kubernetes' StatefulSet resource. Therefore, Acto infers the semantics of cassandraDataVolumeClaimSpec using a structural mapping.

5) Inferring semantics from source code: Acto cannot use property structure to map primitive properties (e.g., integer). Also, naming conventions can be ambiguous or unreliable. For example, the integer size property maps to replicas in Kubernetes' StatefulSet. To map primitive properties, Acto analyzes operator code. The idea is to track the data flow of the property value in the operator code and analyze how the values are used. If a property value is passed to a Kubernetes API or assigned to a Kubernetes resource object, Acto maps the property to a Kubernetes object that stores its value.

Acto implements a static taint analysis to track property values. The initial taints are pointers and references to the desired-state declaration (e.g., cr.spec) and the taints are propagated via data-flow dependencies. The analysis is field sensitive—to track each primitive (sub-)property in the declaration—, inter-procedural and context sensitive.

6) Generating Property Values: To generate values for properties with inferred semantics, Acto currently implements 57 property-specific generators based on Kubernetes resource semantics. Most of these properties are composite. The generators focus on high-level semantics to exercise different scenarios. Each generator creates property values to realize a scenario. We find that most properties exposed by operation interfaces (83% on average in our evaluated operators) can be mapped to Kubernetes resources. Acto's generators are invoked at runtime. Some generators read environment and runtime information to inform value generation (e.g., an unsatisfiable affinity rule).

For properties whose semantics Acto cannot infer, Acto mutates current values based on their data types while satisfying syntactic constraints. Acto only mutates primitive sub-properties of composite properties. Acto's mutation ensures syntactic validity but does not guarantee semantic meaningfulness. Mutated values that are not semantically meaningful help check for vulnerabilities to misoperations. Our manual inspection during Acto evaluation shows that 80+% of mutations are semantically meaningful.

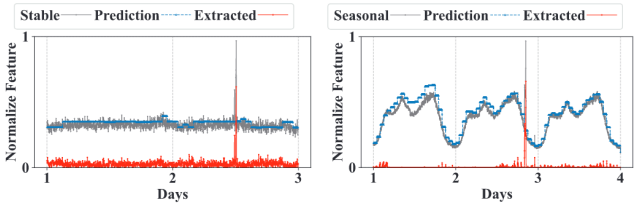
IV. Evaluation

A. AND

1) Offline & Online Processing.: Real-time is the key requirement of the detection algorithm for fast anomaly/failure discovery and recovery. To achieve this goal, AND adopts a hybrid offline and online processing via the low-cost MaxCompute [26] and real-time Flink [16] respectively. The offline stage performs compute-intensive clustering, prediction and model training, while the online stage only extracts real-time features and then detects anomalies.

Offline Clustering. AND uses offline clustering for 'day+1' detection. Partial container IPs may occasionally be assigned to other applications and have different retxs patterns compared with the previous clustering results. The time series of these IPs will be identified and penalized in feature engineering. The clustering results will be updated the next day.

Offline Prediction. AND designs dedicated offline prediction for different clustering of time series (Figure 7). On the one



(a) Stable time series. (b) Seasonal time series.

Fig. 6. Effects of detection procedure.

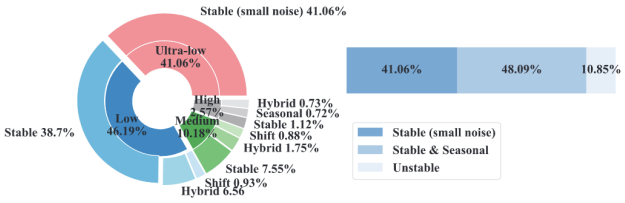


Fig. 7. Clustering of time series (intranet): (1) Stable time series with small 'noise' take 41.06%; (2) Stable and seasonal time series with predictable 'noise' take 48.09%; (2) Unstable time series (mean-shift or hybrid) take 10.85%

hand, partial time series have an ultra-low frequency of retxs. AND adopts a simple difference with constants as features. On the other hand, AND adopts a lightweight predication to eliminate the 'noise' of time series, based on the expectation and variance of history windows.

Offline Training & Online Detection. For offline training, AND randomly samples time series from a one-month period and also includes time series of anomalies/failures to enhance the robustness. The real-time features adopt simple arithmetic calculations using prediction values and penalizing/scaling factors pre-processed offline. Finally, the online detection takes real-time features as inputs and outputs the abnormal IPs and timestamps for further aggregation and diagnosis.

2) Effective Data Filtering.: Next, we demonstrate that the detection procedure effectively extracts features of anomalies and also reduces data volume.

As shown in Figure 6, we intuitively show the effects of the detection procedure, taking stable and seasonal time series as examples. The detection procedure eliminates the 'noise' of time series by lightweight prediction. The features are also penalized and scaled via coefficients in cyclic windows and variance in short-time windows.

B. Hyrax

In this section, we demonstrate that Hyrax can successfully mitigate any VM performance impact of degraded mode operation. For space reasons, we focus on the more complex case of memory performance (memory latency and bandwidth).

1) Server-level experiments.: Figure 8 compares VM memory bandwidth of Hyrax and Naïve on a degraded server to a healthy server. The degraded server has A1 and A2 deactivated. Hyrax allocates the VM using colors 0-3, depending on VM core count. We find that memory bandwidth under

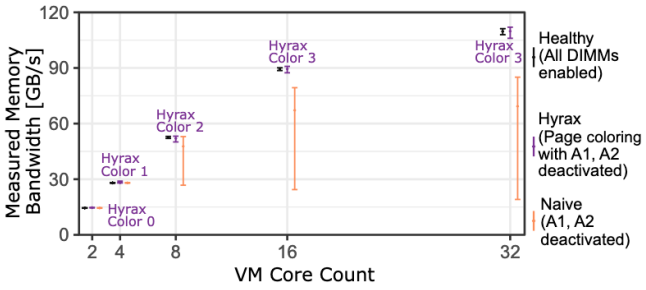


Fig. 8. Peak memory bandwidth of a healthy server, a Hyrax server with page coloring, and a naïve implementation of degraded servers with two DIMMs deactivated.

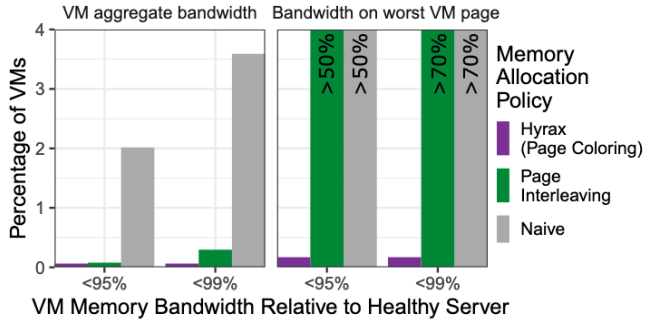


Fig. 9. Hyrax almost always achieves the same VM memory bandwidth on degraded nodes as VMs would on healthy servers.

Hyrax is within 1% of the healthy server. In contrast, Naïve's performance is highly variable with mean bandwidth up to 36% lower and worst-case bandwidth up to 82% lower than on the healthy server.

We also tested memory latency. In all three systems, and across all experiments, the unloaded memory latency reported by MLC for the degraded server remains within 5% of the healthy server.

2) Large-scale page coloring simulations: The previous experiment focused on a single VM in isolation for one particular failure pattern. For a more complete view of VM performance under Hyrax we use simulations that are driven by actual traces of VM arrivals and departures to capture the effect of VM churn and also simulate component deactivation based on real failure traces to capture the rich set of failure patterns that arises in practice. (The traces come from our cluster simulations). We play back these VM events in server-level simulations of the three memory allocation policies: Hyrax page coloring, page interleaving, and Naïve.

Figure 9 shows the percentage of VMs with less than 95% and 99% of the bandwidth of a healthy server, both for VM aggregate bandwidth (left) and bandwidth of the VM's worst page (right). With Hyrax, fewer than 0.16% of VMs see bandwidth on their worst page that is lower than 99% of the worst-page bandwidth achieved on a healthy server. VM aggregate bandwidth under Hyrax is even closer to that of a healthy server.

Table 4. The Kubernetes operators that we evaluate.

Operator	System	Dev.	# Stars	LOC	# E2E Tests
CassOp	Cassandra	K8ssandra	148	23.1K	48
CockroachOp	CockroachDB	Official	238	17.4K	21
KnativeOp	Knative	Official	157	16.3K	7
OCK/RedisOp	Redis	OCK	531	2.5K	0
OFC/MongoOp	MongoDB	Official	977	17.1K	62
PCN/MongoOp	MongoDB	Percona	268	15.0K	31
RabbitMQOp	RabbitMQ	Official	669	14.7K	8
SAH/RedisOp	Redis	Spotahome	1303	10.5K	1
TiDBOp	TiDB	Official	1130	132.8K	131
XtraDBOp	XtraDB	Percona	448	15.5K	37
ZooKeeperOp	ZooKeeper	Pravega	332	5.5K	8

Fig. 10. The Kubernetes operators that we evaluate.

Page interleaving also results in a low percentage of VMs that achieve less than 95-99% of the aggregate memory bandwidth of a healthy server. However, more than half of VMs include at least one memory page with significantly lower bandwidth. We also note that page interleaving increases a VM's page table by orders of magnitude. This leads to a high rate of TLB misses and increased memory access latency. In practice, we know that memory access latency is even more important than bandwidth — internal production workloads lose 5-15% of performance for small page sizes. Thus, interleaving is not practical.

Naïve is compatible with large page sizes but more than 2% of VMs achieve less than 95% of the aggregate bandwidth goal. This grows to 3.5% for a goal of 99% and above 50% when considering the worst page in a VM. While Naïve performs well on average, tail performance matters at scale.

C. Acto

Acto's premise is that fully automatic end-to-end correctness testing for unmodified operators is viable and effective. We answer three research questions: (1) Can Acto effectively find new bugs in real-world operators? (2) How efficient is Acto? (3) Are Acto's signaled alarms trustworthy?

We apply Acto to eleven popular open-source Kubernetes operators which manage nine cloud systems (Figure 10). All evaluated operators are developed by the official teams of the managed systems, or by companies that sell services built around the managed systems.

1) Finding New Bugs and Vulnerabilities: Acto finds previously unknown bugs in all evaluated operators, 56 bugs in total. We reported all these bugs. So far, 42 were confirmed and 30 have been fixed. No bug report was rejected. Acto found all 56 bugs. Acto missed one bug, due to not being able to infer the semantics of a primitive property that is needed to generate a scenario.

Acto generates e2e tests to reproduce all 56 bugs that it detects; developers can add these e2e tests to their regression test suite. In fact, for six bug fixes, developers added regression tests that perform the same state transition generated by Acto. Our experience tells that the generated e2e tests are invaluable for debugging and validating bug fixes.

Many bugs detected by Acto have severe consequences: managed-system failures, reliability issues, and security issues. Estimating the likelihood of encountering each bug "in the field" is hard—the data for such estimation is not publicly available. However, a bug detected by Acto was also encountered by a real user after we reported it. Also, some previously reported bugs are similar to those that Acto detects. Note that the evaluated operators are popular open-source projects, suggesting that operator correctness is hard to achieve.

Acto also detects 630 misoperation vulnerabilities. Each vulnerability corresponds to a unique misoperation that drives the managed system into an error state.

2) Test Efficiency: All experiments are run on Cloudlab Clemson c6420 machines with 2 Intel Xeon Gold 6142 CPUs (16 cores) and 376 GB of memory, with Ubuntu 20.04 LTS. Campaign times vary from 4.72 to 57.51 hours across operators. Using eight machines, test campaigns for all operators finish in less than eight hours. So, Acto can be run nightly.

3) False Positive: Acto's alarms have a low false positive rate. Acto reports no false alarm. Every test failure during the test campaigns points to either a bug in the operator code or a misoperation vulnerability. In total, Acto reports 2243 test failures: 738 test failures are caused by the 56 bugs in the operator and six bugs in Kubernetes and Go runtime, and 1505 test failures are caused by 630 misoperation vulnerabilities. Fixing one bug or vulnerability may resolve multiple test failures. We are automating alarm clustering based on fault localization [13], [38], but it is now beyond the scope of testing.

Acto reports four false alarms in total. It reports 2071 test failures in total; among them, 653 test failures are caused by 55 bugs in operators and six bugs in Kubernetes or Go; 1414 test failures are caused by 616 misoperation vulnerabilities. Therefore, the overall false positive rate of Acto is 0.19%, or 4 out of 2071 alarms. All four false alarms are caused by unsatisfied predicates when Acto changes properties. Acto is unable to infer dependencies that do not follow the naming convention. For example, in ZooKeeperOp, the property, ephemeral, depends on a predicate: another property, storageType, must also be set to "ephemeral". Hence, Acto fails to satisfy the predicate when changing the ephemeral property, but it expects a state change and raises a false alarm. These dependencies are captured by Acto through control-flow analysis.

V. Related Work

A. Active Probing.

Pingmesh [23] targets physical networks in large data centers. VNET Pingmesh [39] and VTrace [40] extend the coverage to virtual networks in the clouds. Zoonet [6] further extends the scope to end hosts, which covers anomalies of vSwitches/NICs and VMs/netdevs via ARP ping. However, the out-band probing may not cover problems of actual service traffic. For example, ACL rules allow probing packets by default and the ARP packets experience different paths with

TCP/IP packets of services at the kernel stack. They target non-transient network failures (no shorter than the probing interval [41]) and intranet traffic (internet is uncontrollable [42]). Last but not least, AND reveals for the first time that the probing systems fall short in assessing the impacts of network anomalies on real application traffic in a cloud-native environment.

B. End-host Monitoring.

To correlate applications with network paths end-to-end, existing works collect connection-, linkand even packet-level metrics at end hosts. PathDump [43] and Facebook [44] propose to correlate connections with the network paths by marking packets at each hop and then parsing packets at end hosts. 007 [2] also collects retxs and queries the network path of each retx via Traceroute. However, Traceroute incurs too much overheads to the switch's control plane with many retxs. NetPoirot [45] identifies root causes of failures only using TCP statistics at one host, which relies on artificial failure injections. In all, collecting fine-grained metrics helps a lot in anomaly diagnostics, however, incurs considerable overheads in long-term operation and should be enabled on-demand.

C. Datacenters that fail-in-place.

Related to our work are the general efforts toward lights-out data centers such as containerized datacenters [32], underwater datacenters [13], and zero-maintenance storage systems [46], [47]. In our evaluation, AoN with high batch repair intervals (12m) represents these approaches. Unfortunately, the loss in availability or cost (hardware, power, space) to make up for this loss is prohibitive without degraded mode.

D. In-network Telemetry.

The programmable data plane and in-network telemetry promote novel monitoring systems at switches/NICs [47]. PINT [35] and NetSeer [32], [48] record network-wide statistics and abnormal events at programmable switches respectively. SpiderMon [30] builds a closed-loop between monitoring and posterior diagnosis to achieve low overhead and high coverage. BufScope [49] monitors request-level anomalies of application RPCs by correlating requests at end hosts (SmartNICs) to network paths (programmable switches). While BufScope [12] has been deployed in Alibaba's production storage application, these solutions rely on new hardware (e.g., programmable switches).

E. Mechanisms to implement fail-in-place.

We borrowed the term degraded mode from RAID systems [19], where upon failure of a drive, the system seamlessly continues to operate until the failed drive is replaced, however at reduced capacity and reduced performance.

There are many existing fault-tolerance approaches that use component-internal redundancy [1], [24], [40], [50]. Hyrax targets the left-over failures not already covered by these approaches. It can be viewed as taking degraded mode to the extreme and applied to even combinations across different devices. As such, Hyrax has different requirements that raises novel challenges.

F. Improving repairs and redundancy.

Recent efforts for reducing the reliance on human technicians in lights-out datacenters explore the use of robots to replace hardware components [51]. Currently, this technology is not sufficiently capable, versatile and economical to be employed at scale. Our work presents a solution that can be deployed immediately in today's systems.

Finally, systems that require no or minimal repairs throughout their lifetime are common in the context of embedded systems, for example, as part of autonomous vehicles, airplanes or satellites [16]. However, these are special purpose systems with specialized components and significant redundancy. In contrast, we are exploring whether a cluster based on commodity data center components can operate with no or minimal repair throughout its lifetime through the use of fail-in-place.

G. Operation errors.

Prior work identified operation errors as major causes of production failures [3], [52], [53]; they result mostly from human mistakes. As human-based operations are increasingly being replaced by automated operation programs, the correctness of those programs is critical. Acto is a first step towards automatic testing of operation correctness.

We believe that Acto's ideas can apply beyond Kubernetes to other cloud platforms like Twine [9], ECS [21], and Borg [38]. These platforms also adopt declarative, statereconciliation patterns for operators or controllers, as a result of many design iterations [54] and discussions [47].

DCM [55] uses declarative programming to synthesize cluster managers based on constraint solving; the idea can potentially be extended for custom operators. However, most operators are currently written in imperative code.

Acto is complementary to prior work on software deployment [11], [38] and configuration [8], [16], [56]. Acto checks programs that perform those operations rather than the correctness of code or configuration changes.

Sieve [54] is a closely related testing technique. It finds bugs in Kubernetes controllers that are triggered by external faults like node failures, network delays, etc. Operators are custom controllers for managing systems atop the Kubernetes platform. Acto is fundamentally different from, but complementary to Sieve. In essence, Sieve is a fault injector that checks fault tolerance, while Acto is an end-to-end test generator that checks functional correctness. Sieve cannot find the bugs Acto detects, because it assumes that the operator works correctly without faults. Sieve detects bugs by comparing operator executions with and without injected faults. Sieve does not report errors in any fault-free reference execution. More importantly, Sieve takes test workloads as input—those test workloads are currently written manually, but it is challenging and costly for developers to write comprehensive test workloads. Acto automatically generates test workloads (i.e.,

"test campaigns" in Acto's terminology). Conversely, Acto cannot directly detect bugs that Sieve finds, because Acto does not inject external faults. We discuss potential Acto and Sieve integration.

VI. Conclusion

In this report, we presents three works that enhance cloud reliability. AND exploits a single metric of TCP retxs to build the unified monitoring and diagnosing capability, which enables minute-level anomaly detection and facilitates fast failure recovery. According to the operational experience of over three years, AND demonstrates its superiorities from several aspects, including impact assessment at (micro)service levels, multiple-problem-domain anomaly routing, extremely low overheads and generalization ability. We also introduce Hyrax, a datacenter stack that enables compute servers with failed components to continue hosting VMs while hiding the underlying degraded capacity and performance. A key enabler of Hyrax is a novel model of changes in memory interleaving when deactivating faulty memory modules. Experiments on cloud production servers show that Hyrax overcomes common hardware failures without impacting peak VM performance. In large-scale simulations with production traces, Hyrax reduces server repair requirements by 50-60% without impacting VM scheduling. For cloud operator bugs, we introduce Acto, an automatic technique for testing cloud-native operators end to end with the managed systems. We show that Acto's state-centric approach enables effective and practical end-to-end testing that is readily applicable to existing operators and complements the significant inadequacy of manually written tests. Our goal now is to make Acto a common utility in developing and testing operators, towards correct automation of cloud system operations.

References

[1] M. H. Asyrofi, Z. Yang, I. N. B. Yusuf, H. J. Kang, F. Thung, and D. Lo, "Biasfinder: Metamorphic test generation to uncover bias for sentiment analysis systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5087–5101, 2021.

[2] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[3] A. Ait, J. L. C. Izquierdo, and J. Cabot, "An empirical study on the survival rate of github projects," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 365–375.

[4] "Github rest api documentation." [Online]. Available: https://docs.github.com/en/rest?apiVersion=2022-11-28

[5] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is it all lost? a study of inactive open source projects," in *Open Source Software: Quality Verification: 9th IFIP WG 2.13 International Conference, OSS 2013, Koper-Capodistria, Slovenia, June 25-28, 2013. Proceedings 9*. Springer, 2013, pp. 61–79.

[6] A. Gelman and J. Hill, *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

[7] R. G. Smart, "Subject selection bias in psychological research." *Canadian Psychologist/Psychologie canadienne*, vol. 7, no. 2, p. 115, 1966.

[8] A. J. Nederhof, "Methods of coping with social desirability bias: A review," *European journal of social psychology*, vol. 15, no. 3, pp. 263–280, 1985.

[9] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on npm," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2208–2223, 2021.

[10] S. He, X. Zhang, P. He, Y. Xu, L. Li, Y. Kang, M. Ma, Y. Wei, Y. Dang, S. Rajmohan *et al.*, "An empirical study of log analysis at microsoft," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1465–1476.

[11] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2020.

[12] J. Jiang, D. Lo, X. Ma, F. Feng, and L. Zhang, "Understanding inactive yet available assignees in github," *Information and Software Technology*, vol. 91, pp. 44–55, 2017.

[13] M. Guizani, T. Zimmermann, A. Sarma, and D. Ford, "Attracting and retaining oss contributors with a maintainer dashboard," in *Proceedings of the 2022 ACM/IEEE 44th International Conference on Software Engineering: Software Engineering in Society*, 2022, pp. 36–40.

[14] Snyk, "Snyk vulnerability database," 2023. [Online]. Available: https://security.snyk.io/

[15] Y. Yu, A. Benlian, and T. Hess, "An empirical study of volunteer members' perceived turnover in open source software projects," in *2012 45th Hawaii International Conference on System Sciences*. IEEE, 2012, pp. 3396–3405.

[16] C. Miller, D. G. Widder, C. Kästner, and B. Vasilescu, "Why do people give up flossing? a study of contributor disengagement in open source," in *Open Source Systems: 15th IFIP WG 2.13 International Conference, OSS 2019, Montreal, QC, Canada, May 26–27, 2019, Proceedings 15*. Springer, 2019, pp. 116–129.

[17] K. Crowston, K. Wei, J. Howison, and A. Wiggins, "Free/libre open-source software development: What we know and what we do not know," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, pp. 1–35, 2008.

[18] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*. IEEE, 2011, pp. 275–284.

[19] "How and when to deprecate apis." [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/deprecation/deprecation.html

[20] GitHub, "Octoverse 2022: The state of open source," 2022. [Online]. Available: https://octoverse.github.com/

[21] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[22] naveen, "How to deprecate a python package?" 2021. [Online]. Available: https://github.com/pypa/packaging.python.org/issues/883

[23] Libraries.io, "Libraries.io open data," 2020. [Online]. Available: https://libraries.io/data

[24] J. Fox and S. Weisberg, *An R Companion to Applied Regression*, 3rd ed. Thousand Oaks CA: Sage, 2019. [Online]. Available: https://socialsciences.mcmaster.ca/jfox/Books/Companion/

[25] "Is pep 541 still the correct solution?" [Online]. Available: https://discuss.python.org/t/is-pep-541-still-the-correct-solution/27436/10

[26] C. Henry, L. University, and U. Berkeley, "Measuring the economic value of open source," 2023. [Online]. Available: https://project.linuxfoundation.org/hubfs/LF%20Research/Measuring%20the%20Economic%20Value%20of%20Open%20Source%20-%20Report.pdf?hsLang=en

[27] K. Blind and T. Schubert, "Estimating the gdp effect of open source software and its complementarities with r&d and patents: evidence and policy implications," *The Journal of Technology Transfer*, pp. 1–26, 2023.

[28] C. Robert, "Pep 508 – dependency specification for python software packages," 2015. [Online]. Available: https://peps.python.org/pep-0508/

[29] H. Li, F. R. Cogo, and C.-P. Bezemer, "An empirical study of yanked releases in the rust package registry," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 437–449, 2022.

[30] H. Fang, H. Lamba, J. Herbsleb, and B. Vasilescu, "" this is damn slick!" estimating the impact of tweets on open source project popularity and new contributors," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2116–2129.

[31] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 266–277.

[32] Y. Xi, L. Shen, Y. Gui, and W. Zhao, "Migrating deprecated api to documented replacement: Patterns and tool," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, 2019, pp. 1–10.

[33] C. Miller, C. Kästner, and B. Vasilescu, ""we feel like we're winging it:" a study on navigating open-source dependency abandonment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1281–1293.

[34] I. Pashchenko, "Decision support of security assessment of software vulnerabilities in industrial practice," Ph.D. dissertation, University of Trento, 2019.

[35] "Pep 541 – package index name retention." [Online]. Available: https://peps.python.org/pep-0541/

[36] IBM, "What is the log4j vulnerability?" 2023. [Online]. Available: https://www.ibm.com/topics/log4j

[37] Microsoft, "Microsoft forms," 2021. [Online]. Available: https://www.microsoft.com/en-us/microsoft-365/online-surveys-polls-quizzes

[38] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 672–684.

[39] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, pp. 1–34, 2021.

[40] P. McCann, "How to deprecate a pypi package," 2020. [Online]. Available: https://www.dampfkraft.com/code/how-to-deprecate-a-pypi-package.html

[41] H. He, Y. Xu, X. Cheng, G. Liang, and M. Zhou, "Migrationadvisor: Recommending library migrations from large-scale open-source data," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2021, pp. 9–12.

[42] Synopsys, "The heartbleed bug," 2020. [Online]. Available: https://heartbleed.com/

[43] D. Qiu, B. Li, and H. Leung, "Understanding the api usage in java," *Information and software technology*, vol. 73, pp. 81–100, 2016.

[44] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of jupyter notebooks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1622–1633.

[45] OpenAI, "Introducing chatgpt," 2022. [Online]. Available: https://openai.com/blog/chatgpt

[46] npm, "npm deprecate," 2022. [Online]. Available: https://docs.npmjs.com/cli/v8/commands/npm-deprecate

[47] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[48] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 233–244.

[49] "Sample size calculator." [Online]. Available: https://www.surveysystem.com/sscalc.htm

[50] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

[51] J. Ruohonen, K. Hjerppe, and K. Rindell, "A large-scale security-oriented static analysis of python packages in pypi," in *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE, 2021, pp. 1–10.

[52] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," in *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, 2010, pp. 301–310.

[53] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," *IEEE Transactions on Software Engineering*, 2023.

[54] P. Gedam, "Adding a mechanism to deprecate a published project," 2022. [Online]. Available: https://discuss.python.org/t/adding-a-mechanism-to-deprecate-a-published-project/13937

[55] S. Overflow, "2023 developer survey," 2023. [Online]. Available: https://survey.stackoverflow.co/2023/#most-popular-technologies-language

[56] PyPI, "Json endpoints reference," 2023. [Online]. Available: https://warehouse.pypa.io/api-reference/json.html