Check before You Change: Preventing Correlated Failures in Service Updates

Ennan Zhai, *Alibaba Group;* Ang Chen, *Rice University;* Ruzica Piskac, *Yale University;* Mahesh Balakrishnan, *Facebook;* Bingchuan Tian, *Nanjing University;* Bo Song and Haoliang Zhang, *Google*

https://www.usenix.org/conference/nsdi20/presentation/zhai

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Check before You Change: Preventing Correlated Failures in Service Updates

Ennan Zhai[†], Ang Chen[‡], Ruzica Piskac[◦], Mahesh Balakrishnan[§,∗]
Bingchuan Tian[♮], Bo Song[•], Haoliang Zhang[•]
[†]*Alibaba Group* [‡]*Rice University* [◦]*Yale University* [§]*Facebook* [♮]*Nanjing University* [•]*Google*

Abstract

The reliability of cloud services can be significantly undermined by correlated failures due to shared service dependencies, even when the services are already replicated across machines. State-of-the-art failure prevention systems can proactively audit a service before its deployment to detect risks for correlated failures, but their auditing speeds are too slow for frequent service updates. This paper presents CloudCanary, a system that can perform *real-time* audits on service updates to identify the root causes of correlated failure risks, and generate improvement plans with increased reliability.

CloudCanary achieves this with two primitives, SNAPAUDIT and DEPBOOSTER. SNAPAUDIT leverages two insights to achieve high accuracy and efficiency: a) service updates typically affect only a small part of the service stack, allowing the majority of previous auditing results to be reused; and b) structural reliability auditing tasks can be reduced to a Boolean satisfiability problem, which can then be solved efficiently using modern SAT solvers. DEPBOOSTER, on the other hand, can generate improvement plans efficiently by reducing the required reasoning load, using novel techniques such as model counting. We demonstrate in our experiments that CloudCanary can perform audits over large deployments 200× faster than state-of-the-art systems, and that it consistently generates high-quality improvement plans within minutes. Moreover, CloudCanary can yield valuable insights over real-world traces collected from production environments.

1 Introduction

High reliability is an essential requirement for cloud services. To enhance reliability, cloud providers typically replicate states and functionality across multiple servers, under the assumption of failure independence [33, 34, 54].

Reality, however, is more complicated. The complex, multi-layered nature of network/software stacks in cloud services may conceal underlying interdependencies between seemingly independent components, such as network switches and software modules. Failures of these common service dependencies can lead to correlated failures *despite replication*, causing service downtime [11, 25, 39, 70]. For example, a faulty top-of-rack (ToR) switch would affect all replicas in the same rack [18], and a buggy software component could propagate failures across all service instances it supports [38]. Such incidents have repeatedly made the headlines: in one of the Rackspace outage events [9], glitches in two core switches caused multiple servers to be inaccessible, leading to significant service disruption; in another incident, a single faulty data collector in Amazon EBS brought down the Relational Database service in an entire availability zone [3].

A number of previous efforts have focused on diagnosing the root causes of correlated failures [14, 23, 47, 58]. While this is useful, post-failure diagnostics typically involves prolonged failure recovery time [12, 64], as even the best of diagnostic tools cannot *prevent* service outages. Such outages can be quite costly: on average, a single datacenter outage can cause an economic loss of $740,357 [4].

More recent proposals aim to proactively prevent correlated failures by auditing the structural reliability of cloud services before deployment [22, 70, 71]. At a high level, these systems collect a comprehensive set of structural dependency data in cloud services, and construct a system-wide fault graph to encode the dependencies. They then identify potential risks for correlated failures from the fault graph.

However, state-of-the-art auditing systems are designed to perform audits at service initialization, not for conducting *real-time* audits throughout the service lifetime. Runtime audits are necessary, because existing work has shown that many dependencies potentially causing correlated failures are introduced by network and software updates (*e.g.*, reconfigurations and upgrades) during service runtime [39]. For example, a Gmail service upgrade configured microservice replicas to share the same vulnerable component, which later rendered user data unavailable for many hours [5]. Existing systems are impractical for real-time audits for two reasons.

- First, they are *too slow* in analyzing cloud-scale deployments in real time. For example, a state-of-the-art system takes ∼35 hours to analyze a 30,528-component service [70], making it only possible to perform a few audits per week. This cannot match the update frequency in today's clouds—for instance, Google reported 58 updates per week, roughly one update every three hours [37].
- Second, these tools can only alert the operator to correlated failure risks, but do not offer further support to find effective *improvement plans*. Thus, the operator needs to either manually reason about improvements to the existing deployment, or use automated tools to generate a plan from scratch [22, 70]. The former is error-prone, and the latter may result in a plan that requires considerable service reconfiguration. Moreover, both are inefficient.

[∗] Work done while at Yale University.

In other words, although the operator may have enough lead time to perform audits at service initialization, the high turnaround time of existing systems prohibits their use in real-time auditing during service runtime.

We present CloudCanary, a system that can efficiently and accurately a) alert the operator to the root causes of correlated failure risks introduced by service updates, and b) generate a set of improved deployment plans with higher reliability. CloudCanary achieves this using two primitives—SnapAudit and DepBooster—to help prevent correlated failures during service runtime in a timely manner.

Contribution #1. SnapAudit (§3) can efficiently and accurately identify root causes for correlated failures in a given service snapshot. The design of SnapAudit addresses two challenges. The first is how to rapidly analyze a fault graph representing the service update snapshot. To address this challenge, we propose an *incremental auditing* algorithm to identify a set of *differential fault graphs*, which represents the "delta" between the service snapshots before and after an update. Based on the insight that service updates usually affect a *small* part of service stacks [37, 49, 60], extracting differential fault graphs enables us to avoid the need to re-analyze the entire fault graph from scratch. Second, although differential fault graphs are already smaller than the overall fault graph, analyzing each of them is still NP-hard and time-consuming [63]. We therefore propose an approach that speeds up the fault graph analysis by transforming a differential fault graph into a Boolean formula, and then solving the formula using a high-performance MinCostSAT solver [35].

Contribution #2. DepBooster (§4), on the other hand, helps the operator improve a risk-prone deployment. It allows the operator to specify a reliability goal (*e.g.*, the failure probability needs to be lower than a certain threshold), and then generates a set of alternative improvement plans that meet the specification. DepBooster also addresses two challenges. First, there are infinitely many potential improvement plans to be checked for their capability to satisfy the specified goal. To overcome this challenge, we utilize *network compression* [17]—a technique that can simplify a datacenter network by collapsing symmetric network structures and slicing away irrelevant parts—to significantly reduce the number of states we need to check. Second, even after compression, it still takes a long time to check whether a candidate deployment meets the specified goal. We further propose a novel algorithm based on model counting [20] for efficient checks.

To the best of our knowledge, CloudCanary is the first practical system capable of preventing correlated failure risks in service updates. We have built a CloudCanary prototype and evaluated it with a set of real-world scenarios (§6). Our results show that SnapAudit can identify correlated failure root causes in a 1,183,360-component service within 8 minutes, 200× faster than the state-of-the-art systems, and that DepBooster can find high-quality improvement plans within minutes.

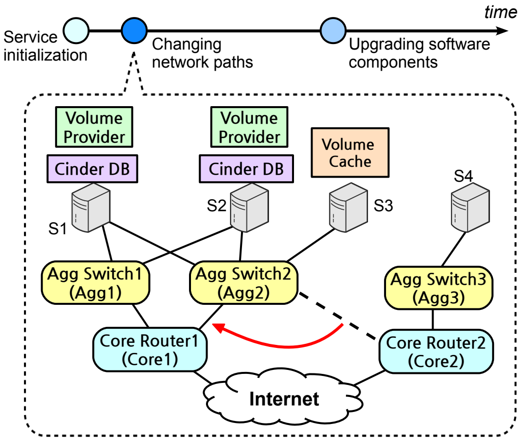


Figure 1: An update that affects the Cinder DB deployment, where the path Agg2→Core2 is shifted to Agg2→Core1 due to an ECMP configuration change.

2 Overview

In this section, we first motivate our problem further (§2.1). Then, we describe the state-of-the-art auditing systems and their limitations (§2.2 and §2.3). Finally, we present the architecture of CloudCanary (§2.4).

2.1 Motivation

Cloud operators ensure service reliability by replicating important state and functionality. Suppose that an operator deploys Cinder DB (a block storage system in OpenStack) in her datacenter, and that she replicates Cinder DB across multiple servers to increase reliability. Unbeknownst to this operator, the replicated Cinder DB instances may share deep dependencies, such as certain network or software components [3, 9]. The failures of such latent common dependencies can lead to a correlated failure across the entire system, undermining the use of replication. Such common dependencies are often called a *risk group*—a (small) number of components whose simultaneous failure results in a correlated failure.

To prevent correlated failures, the operator needs a tool to check for risk groups in a service deployment and generate improvement plans. For instance, if a risk group only contains one element, *e.g.*, a shared switch, it may potentially become a single point of failure. In this case, the operator may want to improve the deployment so that even the smallest risk group contains more than one element. In a similar spirit, if the estimated probability for correlated failures is above a threshold, the operator may want to find a functionally equivalent deployment with a lower failure probability.

Suppose that a service never goes through updates, then the above tasks only need to be performed once at service initialization. However, this is rarely the case in today's clouds, as most services experience frequent updates in their lifetime, and risk groups can be introduced in any of the updates [39]. Figure 1 shows an example: if the network path Agg2→Core2 is shifted to Agg2→Core1 (*e.g.*, due to a change to the ECMP configuration), such an update will introduce a new risk group σ ={Core1}, the fault of which will result in a correlated failure across both Cinder DB instances. Therefore, checking for risk groups and generating improvement plans need to be performed *continuously in real time*.

2.2 Starting Basis: Fault Graphs

Operators already apply a set of "golden standards" for increasing service reliability, such as rack-aware replica placement [8], geo-replication [1], canary tests [10], but achieving a comprehensive understanding of failure risks is a task that needs to be automated. To this end, several state-of-the-art auditing systems [22, 70, 71] have been proposed to proactively check for correlated failures. They do so using a common abstraction called a *fault graph* [63], which represents the structural dependencies of a service.

Fault graph. A fault graph is a layered DAG representing the logical relationships between component faults within a given system [63]. Figure 2 shows the fault graph of the example service in Figure 1. The fault graph has two types of nodes: *fault events* and *logic gates*. The leaf nodes in a fault graph are *basic faults*, which are the smallest units of failures under consideration, *e.g.*, the failure of a switch or software library. The root node in a fault graph represents a *target service fault*, which indicates the failure of the entire service. The rest of the nodes are *intermediate faults*, which describe how basic faults may cause larger service disruptions.

The fault propagation is encoded by layers of *logic gates* in between. If a component fails, the corresponding fault node outputs a 1 to its parent node, which could be either an `AND` or `OR` gate; otherwise the fault node outputs a 0. For an `OR` gate, if any of its children fail, a fault propagates upwards; for an `AND` gate, it only propagates a fault upwards if all of its children fail. Faulty nodes could be further associated with weights that encode the failure probabilities. Each non-leaf node has an *input gate* that connects its lower-layer faults, but leaf nodes, *i.e.*, basic faults, do not have an input gate.

Fault graph generation. State-of-the-art auditing systems (*e.g.*, INDaaS [70], reCloud [22]) have used existing data acquisition tools to automatically collect the structural dependency data needed for generating fault graphs. These tools cover a variety of dependency data, including network path dependencies [56, 70, 77], software component call flows [69, 73–75], and micro-service execution dependencies [24, 61]. Then, these auditing systems invoke various fault graph synthesis algorithms [51, 70–72] to automatically build fault graphs based on the acquired dependency data. Large-scale fault graph generation has been shown to be efficient—for example, INDaaS generates a 70,656-leaf fault graph within minutes [70].

Commercial data centers also deploy a variety of such profiling tools to track inter-service dependency, although the specific tools would differ from company to company. For instance, the Maelstrom [61] system at Facebook collects service dependency data and uses it for failure mitigation. Later in our evaluation, we have also collected dependency data from a production data center using tools that are already in active deployment.

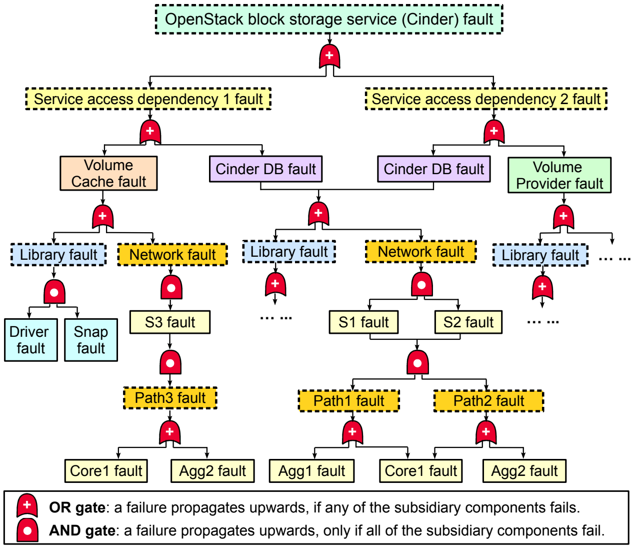


Figure 2: A fault graph representing the post-update service snapshot shown in Figure 1. Path 1, Path 2, and Path 3 represent the links Agg1→Core1, Agg2→Core1, and Agg2→Core1, respectively. Dashed boxes are logical components that do not exist physically.

2.3 State of the Art and Limitations

State-of-the-art systems, such as INDaaS [70], reCloud [22], and RepAudit [71], can perform structural reliability audits on fault graphs to detect risk groups. They then output the identified risk groups to the operator to alert her to the risk. For instance, they may identify {Core1} to be a risk group, because its failure would cause the entire service to fail. However, existing systems all focus on *one-shot* audits at service initialization. They cannot handle *real-time* audits during service runtime due to the following two reasons.

Inefficient risk group auditing. Since detecting risk groups is NP-hard [63], existing auditing systems either perform an exhaustive search, which scales poorly to large deployments, or use heuristics, which sacrifices accuracy. For instance, INDaaS [70] takes ∼35 hours to analyze a 30,528-component service. Such speeds cannot match the frequency of network and software updates in today's clouds—for instance, Google reported 58 network updates per week [37].

Lack of support for generating improvement plans. Existing systems offer no support for the operator to automatically generate improvement plans. As a result, even after performing hours-long audits, the operator still needs to reason about improvement plans if the current service snapshot does not meet her reliability requirements. Existing systems such as INDaaS [70] and reCloud [22] can compute deployment plans from scratch, but such plans may differ considerably from

Table 1: Key techniques in CloudCanary.

Objective	Key techniques	Section(s)
Reusing previous audit results	Caching + Cache refreshes	3.1 + 3.4
Avoiding full-blown Cartesian products	Reduction to DNF (Disjunctive Normal Form) conversion	3.1
Incremental auditing	Differential fault graphs	3.2
Efficient auditing	Reduction to minimum-cost SAT	3.3
Avoiding large-scale Markov chains	Reduction to model counting	4.2
Handling non-uniform probabilities	Adding virtual leaf nodes	4.2
Reducing the search space	Network compression + Search heuristics	4.3

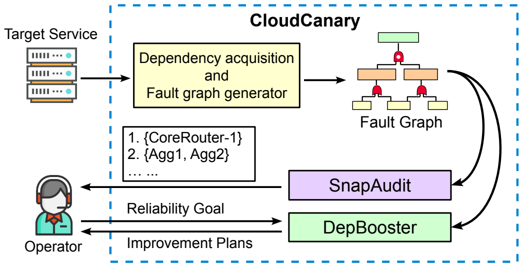


Figure 3: The workflow and architecture of CloudCanary with two novel primitives: SnapAudit and DepBooster.

the current snapshots and require non-trivial reconfiguration. Moreover, these systems are also inefficient to use in service runtime with high update frequency.

Therefore, although the two tasks can be performed with looser time constraints at service initialization, services with frequent updates demand better support for efficient audits and improvements in real time.

2.4 Our Approach: CloudCanary

We propose CloudCanary to achieve the above goals. Figure 3 shows CloudCanary's workflow. For a given service snapshot *S*, CloudCanary collects its dependency data and constructs a fault graph using existing dependency acquisition and fault graph generation modules [70]. The key innovation in CloudCanary is its two primitives SnapAudit and DepBooster. SnapAudit can extract risk groups from *S*, and DepBooster can generate improvement plans, both in a matter of minutes. Table 1 highlights the key techniques we have used and the objectives they are designed to achieve.

SnapAudit. To accelerate auditing, SnapAudit uses two insights. First, since service updates typically just affect a small subset of dependencies [37, 49, 60], there is no need to audit from scratch for each update. Rather, SnapAudit performs a complete fault graph analysis at service initialization, and aggressively reuses cached results to perform *incremental auditing* afterwards. Second, we use a novel encoding to reduce fault graph analysis to a *minimum cost Boolean Satisfiability (SAT) solving* problem, and leverage modern SAT solvers for fast auditing. This insight is driven by the fact that modern SAT solvers can solve complex Boolean formulas efficiently with accuracy guarantees.

DepBooster. The second primitive automatically generates improvement plans to meet a reliability goal, *e.g.*, the minimal risk group containing more than *k* elements, or the failure probability being lower than α. If naïvely done, assessing the failure probability requires solving a long Markov chain [63], and searching through all possible plans further exacerbates the inefficiency. We use a novel reduction to *model counting* to compute the failure probability, as well as a combination of *network compression* and *search heuristics* to reduce the search space.

3 The SnapAudit Design

This section details the design of SnapAudit that identifies the minimal risk groups in a given service snapshot. Figure 4 presents the key algorithms of SnapAudit: FirstAudit is only executed once at service initialization. IncAudit performs incremental auditing for the subsequent snapshots during service runtime. For a given service snapshot, the input of FirstAudit or IncAudit is a fault graph *G* representing its underlying dependency structure, and the output is Σ_G which contains the top-*k* *minimal* risk groups of *G*.

Minimal risk group. A risk group is *minimal* if the removal of any of its constituent elements makes it no longer a risk group. For instance, in Figure 2, there are two minimal risk groups: $\sigma_1 = \{\text{Core1}\}$ and $\sigma_2 = \{\text{Agg1} \wedge \text{Agg2}\}$. On the other hand, $\sigma_3 = \{\text{Agg1} \wedge \text{Core1}\}$ is also a risk group but is not a minimal risk group, because the failure of Core1 alone can cause the entire service to fail. Moreover, we can characterize a risk group's criticality by its cardinality, *e.g.*, σ_1 is more critical than σ_2 because $|\sigma_1| = 1 < |\sigma_2| = 2$—it takes two failures in σ_2 to take down the service but only a single failure in σ_1. The top-*k* risk groups of a given fault graph *G* are a ranked list of minimal risk groups by size or by failure probability. *e.g.*, $\Sigma_G = \{\sigma_1, \sigma_2\}$. Extracting minimal risk groups in a fault graph is NP-hard [63, 72].

3.1 The First Audit

At service initialization, we use FirstAudit to compute the risk groups from scratch. FirstAudit not only audits the overall fault graph *G*, but also *every subgraph* in *G*, thus enabling subsequent audits (performed by IncAudit) to reuse the results for these subgraphs. All audit results are recorded in a key-value cache Σ, where the key corresponds to a particular subgraph, and the value is its top-*k* minimal risk groups. FirstAudit builds a unique identifier for each subgraph by constructing a Merkle Hash Tree [53], and uses the root node's hash as the identifier of the entire subgraph. This allows for a

```
function FIRSTAUDIT($G$)
    if isleaf($G$) then
        $\Sigma_G \leftarrow \{G\}$
    for $c \in G.children$ do
        if $\Sigma_c = \emptyset$ then
            $\Sigma_c \leftarrow$ FIRSTAUDIT($c$)
    $\Sigma_G \leftarrow$ MERGE(G)
    return $\Sigma_G$

function MERGE($G$)
    $c_1, \cdots, c_k \leftarrow G.children$
    if $G.gate = AND$ then
        $\Sigma_G \leftarrow$ DNF($\Sigma_{c_1} \wedge \cdots \wedge \Sigma_{c_k}$)
    else
        $\Sigma_G \leftarrow$ DNF($\Sigma_{c_1} \vee \cdots \vee \Sigma_{c_k}$)
    return $\Sigma_G$

function INCAUDIT($G$)
    $\Pi \leftarrow$ GETBORDER($G$)
    for $t \in \Pi, t.children \notin \Pi$ do
        for $c \in t.children, \Sigma_c = \emptyset$ do
            $\Sigma_c \leftarrow$MINCOSTSAT($c$)
    $\Sigma_G \leftarrow$MERGEALL($G$)
    return $\Sigma_G$

function GETBORDER($G$)
    while $BFS(G)$ with $Q$ do
        $n \leftarrow Q.Pop()$
        if $\Sigma_n = \emptyset$ then
            if $c \in n.child, \Sigma_c \neq \emptyset$ then
                $L.append(n)$
            $Q.Push(n.children)$
    return $L$
```

Figure 4: The key functions in SNAPAUDIT: FIRSTAUDIT and MERGE (§3.1), INCAUDIT and GETBORDER (§3.2).

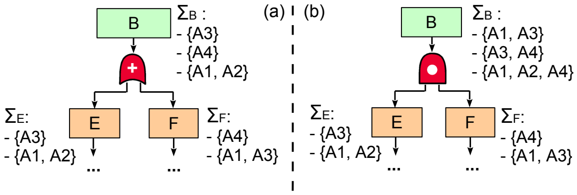


Figure 5: Merging risk groups for AND/OR gates.

more compact encoding of the subgraphs in the cache, given that the number of subgraphs in G is very large. For example, in Figure 5(a), the key of the subgraph rooted at B is $h(B) = h(h(E)||h(F))$, where h is a hash function and $||$ denotes concatenation. Indexing Σ by B's key would return $\Sigma_B = \{\{A3\}, \{A4\}, \{A1 \wedge A2\}\}$.

To generate Σ for both G and its subgraphs, a strawman solution is to directly call existing auditing systems such as INDaaS [70]. However, as discussed in §2.3, these systems are quite slow because their fault graph analysis algorithms scale poorly. Here, any inefficiency would be amplified several times over, because we are computing the minimal risk groups *for each subgraph* in G. To address this problem, we propose a completely different approach to computing the minimal risk groups, using high-performance Boolean formula translation toolchains such as Z3 [27] and Velev [62].

Overall, our FIRSTAUDIT algorithm starts with the leaf nodes, and recursively ascends to upper layers, until it reaches the root node of G. The base case for FIRSTAUDIT is to compute the minimal risk group list Σ_n for a leaf node n, where it simply returns $\Sigma_n = \{\{n\}\}$. In the inductive case, FIRSTAUDIT processes an intermediate node n with children $n_1, \cdots, n_k$ by calling MERGE on n and combining results for all its children. If n's children are connected by an OR gate, we have $\Sigma_n = \Sigma_{n_1} \cup \cdots \cup \Sigma_{n_k}$; otherwise, if n's children are connected by an AND gate we have $\Sigma_n = \Sigma_{n_1} \times \cdots \times \Sigma_{n_k}$, where $\times$ denotes Cartesian product. Figure 5 shows a concrete example.

Reduction to DNF conversion. A naïve MERGE over an AND gate requires a full-blown Cartesian product between risk groups, which leads to state explosion. If the size of each Σ_{n_i} is $|\Sigma|$, merging k of them would result in a set of size $|\Sigma|^k$; after s merges, the size would further grow to $|\Sigma|^{ks}$. To solve this problem, our insight is that MERGE can be achieved by a DNF (Disjunctive Normal Form) conversion, which can be efficiently computed using modern solvers [27]. A Boolean formula is in DNF if it is a disjunction of conjunctive clauses.

Consider the case shown in Figure 5(b), where we have $\Sigma_E = \{\{A3\}, \{A1 \wedge A2\}\}$ and $\Sigma_F = \{\{A4\}, \{A1 \wedge A3\}\}$. We need to compute $\Sigma_B = \Sigma_E \times \Sigma_F$, which can be transformed to a Boolean formula: $\phi = \Sigma_E \wedge \Sigma_F = ((A1 \wedge A2) \vee A3) \wedge ((A1 \wedge A3) \vee A4)$. By using Z3, we can quickly compute the DNF of ϕ, getting $(A1 \wedge A3) \vee (A1 \wedge A2 \wedge A4) \vee (A3 \wedge A4)$. As a result, Σ_B contains three minimal risk groups: $\{A1, A3\}$, $\{A3, A4\}$, and $\{A1, A2, A4\}$. Note that only DNF transformation can output all the minimal risk groups within one-run, and other solvers, *e.g.*, MinCostSAT, do not support such a capability.

3.2 Subsequent Audits

All subsequent audits are performed using INCAUDIT, which reuses the results in Σ generated by FIRSTAUDIT. As shown in Figure 4, INCAUDIT has three steps: GETBORDER, MINCOSTSAT, and MERGEALL. Given a fault graph G, INCAUDIT first uses GETBORDER to identify the differential fault graphs, and then invokes MINCOSTSAT to extract risk groups from each differential fault graph. Finally, INCAUDIT uses MERGEALL to merge the results for the differential fault graphs and those for the unchanged subgraphs, getting the final result Σ_G (*i.e.*, G's minimal risk groups).

GETBORDER. This step identifies a set of special *border nodes* that delineates the changed and unchanged portions of the fault graph. Concretely, a node n with children $n_1, \cdots, n_k$ is called a border node if a) at least one of $n_1 - n_k$'s key has a hit in Σ (*i.e.*, $\Sigma_{n_i} \neq \emptyset$), and b) at least one of them has a miss in Σ (*i.e.*, $\Sigma_{n_j} = \emptyset$). If all n's children have been previously audited, or none of them has been audited, then n is not a border node. For instance, in Figure 6, A and B are border nodes, but C–F are not.

To identify border nodes, we traverse G in a breadth-first order from the root. For each traversed node n, we check whether n has a hit in Σ. If n has a hit, we can reuse its result because 1) n is not a border node, and 2) n's subgraph has not changed. If n misses in Σ (*e.g.*, A in Figure 6), we check its children. If any of n's children has a hit in Σ (*e.g.*, D in Figure 6), we record n as a border node, and recurse and process n's children in order to find more border nodes.

We then extract *differential fault graphs* based on the border nodes and analyze such subgraphs from scratch, starting from the *bottom border node*. A node is a bottom border node if a) it is a border node, and b) none of its subgraphs contains

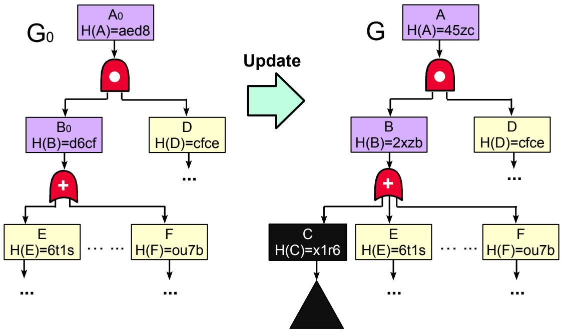


Figure 6: A service update where a subgraph C is added to the fault graph; this changes the keys for subgraphs rooted at A and B. In the updated fault graph, A and B are border nodes, B is the bottom border node, and D–F are unchanged. The subgraph rooted at C is a differential fault graph, which we invoke MINCOSTSAT on to obtain Σ_C. The results for Σ_D, Σ_E, and Σ_F have already been cached in Σ.

more border nodes. For example, in Figure 6, the only bottom border node is B; A is a border node, but not a bottom border node. We identify the bottom border nodes' children who miss in Σ (*e.g.*, C in Figure 6) as the roots of differential fault graphs, and analyze such subgraphs from scratch.

MINCOSTSAT. To analyze a differential fault graph, G_Δ, from scratch, a straightforward approach is to directly invoke FIRSTAUDIT. However, unlike the first audit at service initialization, which could be performed at leisure, INCAUDIT is frequently invoked during service runtime; thus, efficiency is much more critical. Thus, rather than audit all subgraphs in G_Δ, we only audit G_Δ itself. We achieve this by reducing this single audit to a *minimum-cost SAT* problem, which can be efficiently solved using modern SAT solvers. This step is denoted by MINCOSTSAT. For example, in Figure 6, because the subgraph rooted at C is a differential fault graph, we invoke MINCOSTSAT to compute its minimal risk groups, *i.e.*, Σ_C. We detail this MINCOSTSAT reduction in §3.3.

MERGEALL. After we use MINCOSTSAT to compute the risk groups for all differential fault graphs, we need to recompute the risk groups of G. Our insight is that we already have results for the siblings of these differential fault graphs (*e.g.*, D, E, and F in Figure 6), and we could directly use the DNF conversion in MERGE (§3.1), to obtain the risk groups for the entire G. Specifically, we generate a Boolean formula ϕ by only combining all the border nodes' children using their respective logic gates. Then, we transform ϕ into DNF, obtaining Σ_G. For example, in Figure 6, we first generate $\phi = (\Sigma_C \vee \Sigma_E \vee \Sigma_F) \wedge \Sigma_D$, and then transform it into DNF, getting the recomputed Σ_A.

3.3 The MinCostSat Solving

We now detail the design of MinCostSAT function, which can efficiently and accurately extract minimal risk groups from a given fault graph.

At a high level, a minimum-cost SAT problem [35] takes as input a Boolean formula ϕ with n Boolean variables $b_1, b_2, \ldots, b_n$, and a cost vector $\{w_i | w_i \geq 0, 1 \leq i \leq n\}$. The goal is to find a satisfying assignment to these variables such that ϕ evaluates to `True`, and simultaneously minimizing the following value: $W = \sum_{i=1}^{n} w_i b_i$.

We design the MINCOSTSAT function to compute the top-k risk groups. Initially, we transform an input fault graph into a Boolean formula ϕ, and initialize the cost of all the Boolean variables to one. For example in Figure 7, the fault graph at the left-hand can be transformed into (Agg1 $\vee$ Core1) $\wedge$ (Core1 $\vee$ Agg2). We then use a MinCostSAT solver to find the top-k critical risk groups through k rounds. Without loss of generality, for the i-th round, we identify the i-th smallest risk group in three steps: 1) we input the current formula ϕ_i and its cost vector into the MinCostSAT solver to generate the satisfying assignment with the minimal cost, 2) we obtain a risk group by extracting all the `True` literals from the resulting assignment, denoted as ψ. and 3) we use a conjunction to connect the current ϕ_i and the negation of ψ, generating a new $\phi_{i+1} = \phi_i \wedge \neg\psi$ for the next round.

3.4 Further Speedups

Since SNAPAUDIT heavily relies on the cache Σ for efficient audits, we propose two additional techniques to achieve further speedups. First, the results obtained during INCAUDIT can also be cached in Σ, so that Σ would grow over the service lifetime and the hit rate would improve. Second, INCAUDIT does not audit the subgraphs of a G_Δ, but the subgraphs may be needed for subsequent audits. We therefore run a background process that periodically invokes FIRSTAUDIT over the more recent snapshot to refresh the cache. However, we have omitted these techniques from the pseudocode for brevity.

4 The DEPBOOSTER Design

Identifying risk groups is a useful first step, but the operator still needs to reason about ways to increase the service reliability. Rather than ask the operator to achieve this manually, CloudCanary offers a second primitive, DEPBOOSTER, to generate improvement plans in an automated fashion.

4.1 The DEPBOOSTER Workflow

DEPBOOSTER offers the operator an interface to specify "reliability goals", and assesses if the current deployment meets the goals. If not, DEPBOOSTER generates improvement plans with increased reliability. These goals are specified as $spec = req \wedge action \wedge cons$. req is a requirement parameter. It can be a) $rcg > t$, which means the smallest risk groups in the deployment should contain more than t elements, b) $fp < \alpha$, which means the failure probability should be lower than some threshold α, or c) a combination of both. While DEPBOOSTER currently only supports constraints like failure probability and the size of risk groups, more constraints, *e.g.*, key paths, are easily added.

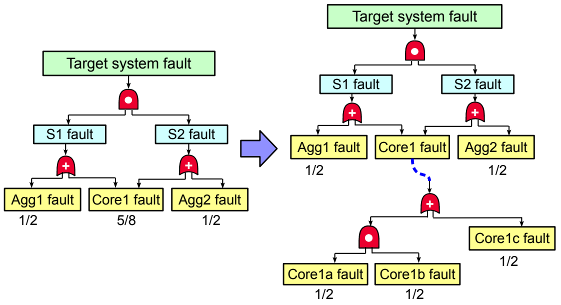


Figure 7: Transforming G by adding virtual leaf nodes.

DEPBOOSTER first assesses whether $rcg > t$ and $fp < \alpha$ already hold on the current snapshot. (Computing whether $rcg > t$ is achieved using SNAPAUDIT, which we described in §3; we defer the algorithm for computing whether $fp < \alpha$ to §4.2.) If both predicates hold, DEPBOOSTER reports so and terminates. Otherwise, it uses the strategies in *action* and the constraints in *cons* to generate improvement plans. *action* specifies an extensible set of basic actions to generate improvement plans with. Currently, DEPBOOSTER supports 1) `mov{r, A→B}`, which moves a service replica `r` from a node `A` to another node `B`; 2) `add{r, A}`, which instantiates an additional replica `r` on node `A`; and 3) `link{A, B}`, which adds a network link between network components A and B. DEPBOOSTER then performs a search for improvement plans based on the basic actions. *cons* contains positive and negative constraints which specify that certain components must or must not be used in an improvement plan.

Example. We now provide a concrete example based on the scenario in Figure 1. Here, the operator provides DEPBOOSTER with a goal: $spec = \{rcg > 1 \wedge fp < 0.08\} \wedge \{mov\} \wedge \{Agg3\}$, which specifies that a) the smallest risk groups should contain more than one elements, and b) the failure probability should be lower than 0.08. If the current snapshot does not meet either of these two goals, DEPBOOSTER will generate a set of improvement plans. Moreover, the *spec* requires DEPBOOSTER to generate improvement plans by only moving replicated instances from the current replica servers to other servers. Finally, any improvement plan must still use the switch Agg3, as specified in *cons*. For this specification, DEPBOOSTER has generated two potential plans: a) `mov{CinderDB, S1->S4}`, and b) `mov{CinderDB, S2->S4}`. In other words, if we migrate the Cinder DB instance on S1 or S2 to S4, then new deployment would meet the desired goals.

4.2 Computing Failure Probability

We now describe how DEPBOOSTER computes the failure probability of a service snapshot. A strawman solution would be to derive the failure probability of the root node from these of the leaf nodes step by step, which is equivalent to computing the conditional probability of a Markov chain [63]. As we will show later, this is a time-consuming operation over large deployments, infeasible to be performed in real time. Instead, DEPBOOSTER uses two techniques to address this.

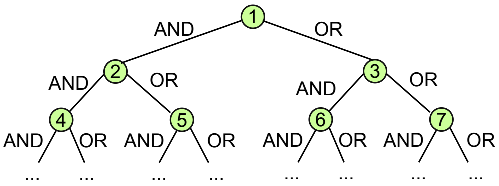


Figure 8: DEPBOOSTER searches through combinations of `AND`/`OR` gates to approximate a given probability.

Technique #1: Model counters. DEPBOOSTER sidesteps the need for Markov chain computation by encoding this into a *model counting* problem. Suppose that the Boolean formula of the fault graph G is ϕ. A model counter [20] can find M—the number of satisfying assignments of ϕ. Assuming for now that all leaf nodes have a failure probability of exactly $\frac{1}{2}$, then the failure probability of G is simply $M/2^n$, where n is the number of leaf nodes in G. Since model counting does *not* need to compute the solutions themselves, but only the number of satisfying assignments, this is much more efficient than solving a Markov chain.

Technique #2: Virtual leaf nodes. However, another challenge arises: in practice, not all leaf nodes have the same failure probability, and such a probability is typically much lower than $\frac{1}{2}$. We address this by adding "virtual nodes" in the fault graph and reducing the problem again into the plain version of model counting. At a high level, we achieve this by substituting a node with failure probability of p with a subtree of virtual nodes, where all virtual nodes have a failure probability of $\frac{1}{2}$, and the failure probability of the entire virtual subtree evaluates to p with a user-defined precision ε. For instance, in the example shown in Figure 7, our goal would be to transform the node with $p = \frac{5}{8}$ (*i.e.*, `Core1 fault`) into a virtual subtree.

Figure 8 shows the solution space that DEPBOOSTER searches through to find a combination of gates that approximates a given failure probability. At any point in the search, the path from the root to a node n represents the current combination of gates. These gates further connect virtual nodes of failure probability $\frac{1}{2}$ (not shown in the figure). For instance, the path from the root to node 5 consists of an `AND` gate and then an `OR` gate, so the formula would be `(v1 AND v2) OR v3`, where `v1`-`v3` are virtual nodes with a failure probability of $\frac{1}{2}$. The failure probability of n_5 can then be computed as $p(n_5) = ((\frac{1}{2} \times \frac{1}{2}) + \frac{1}{2}) - (\frac{1}{2} \times \frac{1}{2}) \times \frac{1}{2} = \frac{5}{8}$. Our algorithm performs a BFS over the solution space, and constructs a combination of gates based on the path from the root to the current node. If $|p(n_j) - p| < \varepsilon$ holds for the current node, the search stops and we use the current combination to approximate a given probability. This transformation converts the fault graph G to a larger fault graph G' with (roughly) the same

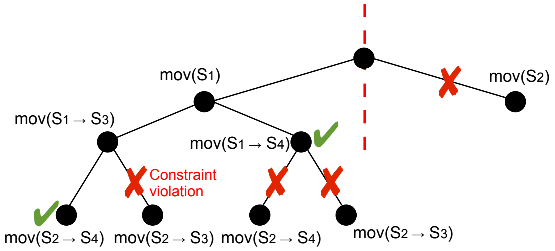


Figure 9: An example state-space tree.

failure probability that can be solved by model counters—*i.e.*, $p = M'/2^{n'}$, where M' is the model counter output for G' and n' is the number of leaf nodes in G'.

4.3 The DepBooster Algorithm

If the current deployment already meets the reliability goals, DepBooster directly terminates. Otherwise, it generates improvement plans by searching through a state-space tree. Each node in this tree represents one concrete move in *action*, and a path from the root to a leaf represents an improvement plan. Since there could be a large number of possible improvement plans, DepBooster uses two techniques to accelerate the search. Below, we use Figure 9 as an example to illustrate how DepBooster generates improvement plans for our running example in §4.1.

Technique #3: Network compression. We use the observation that datacenter network topologies tend to be highly symmetric, and can be "simplified" to equivalent topologies much smaller in size [17]. This enables DepBooster to perform the search on the smaller networks, and then map the solution back onto the original topologies. Driven by this observation, DepBooster transforms the input network topology D to a simplified topology d while preserving its original connectivity and reachability. Briefly, this is achieved by collapsing symmetric network structures (*i.e.*, routers and paths) and slicing away irrelevant structures. For instance, Figure 9 shows how the symmetric branch at S_2 has been pruned. We refer interested readers to the original paper [17] for proofs.

Technique #4: Iterative deepening. DepBooster then generates the state-space tree T_d based on the simplified topology d. It never materializes T_d in its entirety, but only explores it step by step. Concretely, DepBooster performs an Iterative Deepening Depth-First Search (IDDFS) [46] on T_d starting from the root. For each traversed node n, DepBooster checks whether n or any of n's children violates the specified constraints. If any constraint is violated, then the corresponding branches are pruned. For example, in Figure 9, the branch `mov(S2 -> S3)` under `mov(S1 -> S3)` is pruned because moving Cinder DB instances on S1 and S2 to S3 violates the constraint that Agg3 must be used in the new deployment. For the remaining nodes, DepBooster runs IncAudit and the failure probability computation approach (designed in §4.2) to check whether the size of risk groups and failure probability meet the specified goals. For instance, in Figure 9, we do not need to check any branches below the state `mov(S1 -> S4)`, since moving the Cinder DB instance on S1 to S4 has already satisfied the specified goals.

DepBooster can be configured to a) produce improvement plans with the smallest number of actions, b) find the first t improvement plans, and c) run until a timeout occurs. In the running example, we have used b) to find four improvement plans: 1) `mov{CinderDB, S1->S4}`, 2) `mov{CinderDB, S2->S4}`, 3) `mov{CinderDB, S1->S3}`, `mov{CinderDB, S2->S4}`, and 4) `mov{CinderDB, S2->S3}`, `mov{CinderDB, S1->S4}`. The first two plans correspond to those shown in §4.1.

5 Limitations and Discussions

We discuss three high-level limitations of CloudCanary and potential ways to address them.

Quality of inputs. CloudCanary takes two types of inputs as given: a) dependencies, and b) failure probabilities; so it would be limited by the accuracy of the inputs (see §6.5 for a concrete example). For instance, an operator might not know that two upstream ISPs share the same undersea fiber, and that a fiber cut would bring down both networks; or an operator's estimate of the failure probabilities might not be perfectly accurate. In such cases, CloudCanary cannot automatically identify these inaccuracies. However, CloudCanary can benefit from advances in dependency collection systems or failure estimation algorithms: enhancement to CloudCanary's inputs always leads to improved utility.

Dependency granularity. CloudCanary is also limited by the dependency granularity of its data acquisition system; it currently cannot reason about more fine-grained dependencies such as configuration files. If a misconfigured component handles two upper-layer services differently, the current version of CloudCanary would not be able to identify that. This is somewhat akin to the previous limitation, and could benefit from a similar solution—*e.g.*, enhancing the fault graphs to capture configuration files.

Non-deterministic failures. The logic gates in CloudCanary's fault graph are deterministic, which assumes that if two services depend on a common component, the failure of the component would affect both services. This assumption does not capture well non-deterministic and/or partial failures, *e.g.*, when a bit flip in switch TCAM only affects a subset of services but not others. Modeling such behaviors might require extensions to the fault graph abstraction, which we leave as future work.

6 Evaluation

Our evaluation aims to answer three high-level questions: (1) How efficient and accurate is CloudCanary in identifying the risk groups? (2) How quickly can CloudCanary generate improvement plans? and (3) How well can CloudCanary shed light on failure risks in real-world traces?

Table 2: The configuration of our deployments.

	Deploy. A	Deploy. B	Deploy. C
# Switch ports	24	64	128
# Core routers	144	1,024	4,096
# Agg switches	288	2,048	8,192
# ToR switches	288	2,048	8,192
# Virtual machines	3,456	65,536	524,288
# Libraries/Microservices	4,492	79,824	638,592
Total # of components	8,668	150,480	1,183,360

Prototype implementation. We have developed a CloudCanary prototype using a mix of C++, Python, and open-source software libraries. Our system consists of three components: a) fault graph generator, b) SnapAudit, and c) DepBooster. The fault graph generator uses NSDMiner [56] and TS [24] to acquire network and software dependency data, and uses INDaaS [70] to parse and generate fault graphs. Our SnapAudit prototype uses a) a high-performance MinCostSAT solver, Maxino [6], for solving the Boolean formulas that encode the fault graphs, b) the Z3 solver [27] for DNF conversion, and c) a fault graph parser based on pyeda [7] to optimize the encoding and transformation of formulas. Our DepBooster prototype uses a scalable open-source SAT model counter, ApproxMC [2], to compute failure probabilities.

6.1 Experimental Setup

We have emulated a datacenter network with a Clos topology [59], and installed Apache Hadoop 3.2.0 and ZooKeeper 3.4.0 as the cloud services. In the performance experiments, we varied the service size from 8,668 to 1,183,360 software and network components using up to 524 k virtual machines, as shown in Table 2. We also used a real failure probability distribution trace for network devices in our experiments. All machines have an Intel Xeon E5-1620 v2 Quad Core HT 3.7 GHz CPU and 16 GB memory.

Baseline systems. Table 3 presents the three state-of-the-art auditing systems that we have used as the baseline to compare CloudCanary against. Among these systems, INDaaS [70] is more accurate than RepAudit [71] and reCloud [22], but the latter two are faster. This is because the minimal risk group algorithm in INDaaS relies on an exhaustive search, which can produce 100% accurate results but scales poorly. RepAudit and reCloud trade accuracy for efficiency: the former uses a simple MaxSAT solving that cannot guarantee that the identified risk groups are minimal[1], and the latter uses sampling for approximation, which may miss risk groups. Furthermore, we have included a fourth baseline that we call ProbINDaaS [70], which is a randomized version of INDaaS that also relies on sampling for efficiency. We note that reCloud uses a more advanced sampling algorithm (*i.e.*, dagger sampling) than ProbINDaaS (*i.e.*, Monte Carlo), and that reCloud can additionally provide the ability to generate a deployment from scratch to meet a reliability goal. Unlike all these baseline systems, CloudCanary can generate *improvement plans* based on the current deployment, and it performs *incremental auditing while preserving accuracy*. As shown later, CloudCanary achieves 100% accuracy while outperforming all baselines.

Table 3: All evaluated systems and their comparisons.

System	Accurate?	Efficient?	Imp. Plans?
INDaaS [70]	✓	×	×
ProbINDaaS [70]	×	✓✗	×
reCloud [22]	×	✓✗	×
RepAudit [71]	✓✗	✓✗	×
CloudCanary	✓	✓	✓

6.2 Performance: SnapAudit

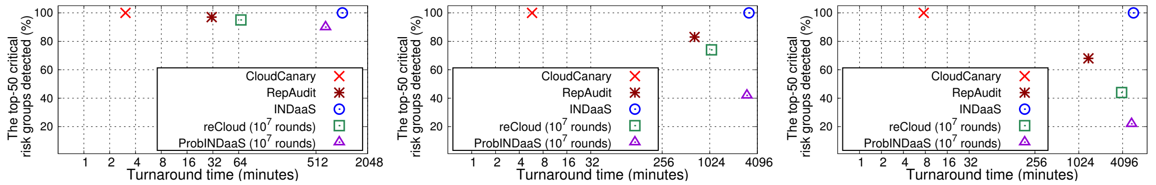
We start by evaluating the performance of SnapAudit using the deployments in Table 2 (A: small, B: medium, C: large). For each deployment, we measured a) the time each system took to audit the service from scratch, and b) the time to audit an updated snapshot when 10% of the hosts and links have been affected. All audits asked for top-50 risk groups.

Efficiency. At service initiation, we observe that INDaaS is the slowest, taking up to ∼5811 minutes on the largest deployment. The two probabilistic approaches ProbINDaaS and reCloud (both with 10^7 sampling rounds) also perform poorly due to the large search space. RepAudit outperforms other baselines and is slightly (∼1.3×) faster than SnapAudit's FirstAudit. However, this is expected, because RepAudit only audits the overall fault graph, whereas FirstAudit audits *both* the overall fault graph *and* its subgraphs to create reusable results for subsequent audits.

We then updated the three deployments by randomly adding or removing 10% hosts and links, and ran the four auditing systems on the resulting deployments A′–C′. Figure 10 shows the turnaround time (X-axis) versus audit accuracy (Y-axis). As we can see, SnapAudit's IncAudit consistently outperforms INDaaS, ProbINDaas, reCloud, and RepAudit for all subsequent audits. On deployment B, IncAudit is faster than the second fastest system RepAudit by 200×.

Accuracy. Moreover, SnapAudit always has an accuracy of 100% across deployments—the same with INDaaS—but RepAudit only has 96%, 83%, and 68% in deployment A'–C', respectively. ProbINDaaS and reCloud are even less accurate. Here, an inaccurate audit in RepAudit, reCloud, and ProbINDaaS means that a) some risk groups are missing from the output, and b) some risk groups generated by these systems are not minimal. For instance, SnapAudit outputs $\Sigma = \{\{A\}, \{B\}, \{C, D\}\}$ as the top-3 risk groups. An inaccurate system, however, may output $\Sigma' = \{\{A, E\}, \{C, D, E\}, \{C, D, F\}\}$, where $\{B\}$ is missing and the rest of the risk groups are not minimal. Therefore, even a low inaccuracy rate (*e.g.*, 100%−96%=4%) means that human operators need to *manually* inspect the results to identify re-

[1] MaxSAT solving means: given an SAT formula with weight one to each clause, find truth values for its variables that maximize the combined weight of the satisfied clauses.



(a) Deployment A'.

(b) Deployment B'.

(c) Deployment C'.

Figure 10: Performance evaluation of CloudCanary, INDaaS, reCloud (with 10^7 rounds of sampling) and ProbINDaaS (with 10^7 rounds of sampling), and RepAudit in one of the update snapshots.

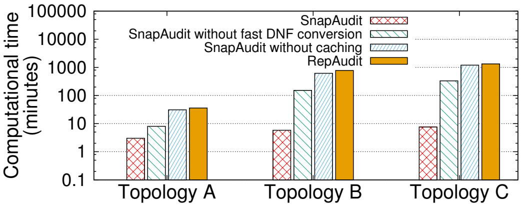


Figure 11: SNAPAUDIT microbenchmarks.

dundancy and reason about the possibility of unidentified risk groups—a task that is time-consuming to perform at runtime.

Microbenchmarks. To further understand the performance improvements of the incremental auditing algorithm in SNAPAUDIT, we have performed a set of microbenchmarks to break down the speedups. We used RepAudit as the baseline, as it performs faster than other systems and is closest to SNAPAUDIT in its use of SAT solvers. For each of the deployments A'–C', we measured the execution times for four scenarios: a) SNAPAUDIT with all optimizations turned on, b) SNAPAUDIT without the fast DNF conversion, c) SNAPAUDIT further without caching previous results, and d) RepAudit. Figure 11 shows that, without any optimization, SNAPAUDIT performs very similarly with RepAudit; the slight speedup comes from the differences in the SAT formulations. The fast DNF conversion led to speedups of 2×–40×, and reusing cached results led to speedups of 4×–8×. These results demonstrate that the optimization techniques in SNAPAUDIT can significantly accelerate incremental auditing.

Degrees of update. A third observation is that the time INDaaS, ProbINDaas, reCloud, and RepAudit took on each subsequent audit is roughly the same with that on their first audits, because they perform each audit from scratch. SNAPAUDIT's INCAUDIT, on the other hand, is significantly faster on subsequent audits than its first audit.

To further evaluate how the degree of updates affects the auditing time of SNAPAUDIT, we tested updates that affect 10%–50% of the components in deployment C, and used SNAPAUDIT to audit these five updates. As shown in Figure 12, the turnaround time of CloudCanary increases roughly linearly with the update percentage. This is good news, because a complete overhaul of a deployed service is rare. Most updates only affect a small part of the service, and they can reap the benefits of CloudCanary easily. On the contrary, since RepAudit never used any incremental algorithm, the RepAudit performance in Figure 10 reflects the update that affects the majority of the deployment.

6.3 Performance: DEPBOOSTER

We now evaluate the performance of DEPBOOSTER for computing failure probabilities and generating improvement plans. For the first task, our baseline systems are INDaaS and RepAudit, both of which solve a Markov chain to obtain the probability. For the second task, our baseline systems are reCloud and RepAudit, although they are not designed to generate improvement plans directly.

Failure probability computation. Figure 13 shows the time DEPBOOSTER, reCloud (with 10^7 sampling rounds) and the baseline system (Markov chain computation) took to compute the failure probability of each deployment. For DEPBOOSTER, we set the precision per leaf node to be 10^{-4} (defined in §4.2) when adding virtual leaf nodes. As shown in Figure 13, DEPBOOSTER achieves a speedup of two to three orders of magnitude compared to reCloud and the baseline. On the largest deployment, DEPBOOSTER only took 2.5 minutes, whereas the baseline and reCloud took more than 10 hours. In terms of the failure probability precision, we found that DEPBOOSTER approximates the probability of the baseline system (which does not use any approximation) with an error of 10^{-3} for all tested deployments, whereas the error in reCloud is 10^{-2} for all tested deployments.

Improvement plan generation. Next, we evaluate the performance of DEPBOOSTER, using reCloud as the baseline. Each deployment hosted the service instances on 50% of the servers, and the query asked for improvement plans to reduce the failure probability to under 0.008 using the `mov` strategy. Figure 14 shows the results. We can see that DEPBOOSTER finished within 30 minutes across deployments, and outperforms reCloud and RepAudit by at least one order of magnitude. DEPBOOSTER can do better for two reasons. First, the model counter-based failure probability computation (§4.2) speeds up the result checking for each candidate solution. In fact, Figure 13 can also be looked as the microbenchmark evaluation for DEPBOOSTER, because the bottleneck operation of DEPBOOSTER is failure probability computation. Second, the pruning technique (§4.3) reduces the number of solutions searched; thus, we can observe few failure probability computations are needed.

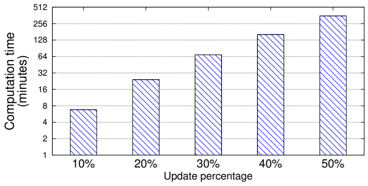


Figure 12: Turnaround time on different degrees of update.

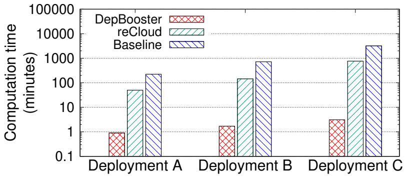


Figure 13: Performance: computing failure probability.

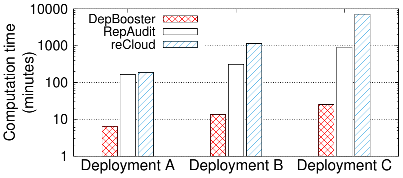


Figure 14: Performance: generating improvement plans.

6.4 Case Study

To better understand how the auditing (in)efficiency affects real-time updates, we have performed a case study that emulates a series of network and software updates. They consisted of 52 updates over the span of one week—we collected the update frequency and distribution from a large-scale cloud provider. We adapted six of these updates from realistic update scenarios [37, 48, 57] and from the Apache issue tracker [13]. All other updates were randomly generated and each of them affected 10% nodes in the deployment. This set of experiments was conducted over a deployment with 576 64-port core routers, 1,152 64-port aggregation switches, 1,152 64-port top-of-rack switches, and 27,648 servers.

Figure 15 shows the results for the first six updates, which we adapted from existing work.

- **Snapshot S0.** At service initialization, the operator set up the entire service, and performed an audit from scratch using the four auditing systems.
- **Snapshot S1: Small updates [48].** The first update changed 1% of network links.
- **Snapshot S2: Large updates [57].** The second update changed 20% servers and 20% links based on a network update trace, and it is designed as "pressure test".
- **Snapshots S3 and S4: Frequent updates [37].** The subsequent two updates occurred within a short interval of seven hours, designed as another pressure test.
- **Snapshots S5 and S6: Software version updates.** The final two updates were to software dependencies, where ZooKeeper was updated from version 3.4.0 to 3.4.6, and then to 3.4.8. They were designed to test the systems' ability to identify software-level risk groups.

Identifying risk groups. Figure 15 shows that existing systems are too inefficient for real-time auditing. INDaaS was only able to finish the auditing for S0 (at service initialization) and S6, but failed for all other updates, because its turnaround time exceeded the intervals between them. RepAudit and reCloud took roughly eight and sixteen hours per snapshot, and they finished S0, S2, and S6, which happened to be spaced out from their previous updates by more than sixteen hours, But they failed to finish for S3–S5, which came close to each other. Recall that, as explained in §6.2, RepAudit and reCloud achieve this speedup by trading accuracy for efficiency, so operators still need to manually reason about missing and non-minimal risk groups after audits.

CloudCanary achieves 100% accuracy in all tested cases, outputting the same results with INDaaS on scenarios where INDaaS was able to finish. However, for each real-time audit, it only took 4.7–6.5 minutes, outperforming INDaaS by 290×, reCloud by 250×, and RepAudit by 150×–200×. The only case where CloudCanary was slightly slower than RepAudit was at service initialization—when auditing S0, CloudCanary needs to audit *all subgraphs* in the fault graph from scratch.

Generating improvement plans. We then ran DepBooster to generate improvement plans for each snapshot. Since the strategies in CloudCanary do not involve changes to software components, we only evaluated S0–S4, where the reliability can be improved using CloudCanary's `mov` strategy. Our reliability goal was specified as $spec = \{rcg > 5 \wedge fp < 0.008\} \wedge \{mov\}$, and we assigned the failure probability of each switch or server to be 0.002 [36]. For S0–S4, CloudCanary generated improvement plans in 4.87, 2.32, 5.77, 7.12 and 6.29 minutes, respectively. In all cases, CloudCanary finished well before the next update arrived. Furthermore, in order to test the effectiveness of our improvement, we injected errors via a chaos-monkey-like way, randomly killing four components, because our constraints set $rcg > 5$. We observed that the improved deployments never failed.

Overall success rates: We now report results for all 52 update snapshots. Our metric is the *success rate* of a system, defined as $r = \frac{m}{n}$, where m is the number of updates for which the system finished on time, and n is the total number of updates. In other words, $m - n$ is the number of updates that cannot be handled due to an audit system's high turnaround time. Overall, INDaaS failed on almost all cases ($r = 1.92\%$) due to its inefficiency. reCloud and RepAudit are faster, but still only had a success rate of 3.85%. CloudCanary, on the other hand, achieved a success rate of 100%, finishing all audits with an average turnaround time of 5.23 minutes.

6.5 Identifying Real Risk Groups

Finally, we evaluate the usability of CloudCanary using a real-world update trace collected from a major service provider. The trace contains 300+ updates to its infrastructure, including software microservices, power sources, and network switches. The operators that executed these updates have already been trained with best practices for service reliability, but systematically understanding service dependencies is always a challenging task. For each update in this trace, we have used CloudCanary to identify the top-5 risk groups, and obtained

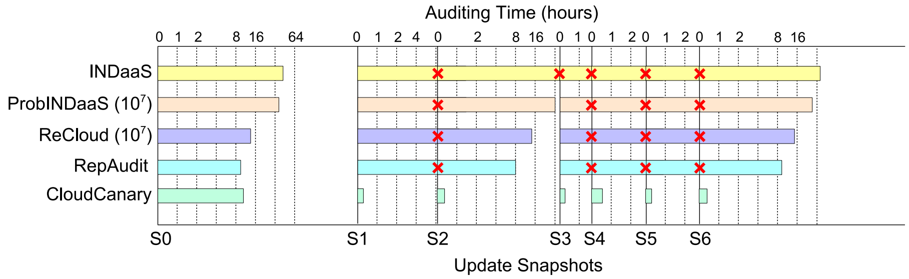


Figure 15: Results on a deployment running Hadoop 3.2.0 and ZooKeeper 3.4.0 in a Clos-topology datacenter with 27,648 hosts and 880 routers. S_i are service updates, which potentially lead to new risk groups.

feedback from the operators. Upon their request, the numbers below are presented as 50+, 10+, and so on, by rounding off their last digits. A key highlight here is that operators have confirmed that 50%+ of these risk groups were previously unknown to them, and that some of them actually caused service downtime in the past.

Microservices. We found 50+ risk groups in the microservice updates. The operators have confirmed that 96% of them could lead to correlated failures; the rest 4% are due to false positives of the dependency collection tool (see §5 for discussion on quality of inputs). One particularly risky example from the operators' feedback is an update that routes all requests to the same authentication service on a single machine. If this machine fails, this would lead to a major outage. The operators can fix this risk group by replicating the authentication service across multiple machines.

Power sources. We found 10+ risk groups in the power sources. Operators have confirmed that all of them could lead to correlated failures, and, in fact, 30%+ of them did trigger service downtime in the past. As a highlight, one of the updates assigned primary and backup power sources in the same cluster to serve several racks hosting a critical service. The cloud provider had experienced multiple hours of downtime due to a failure of these power sources.

Network. CloudCanary reported 30+ risk groups, including ToR/aggregation switches and shared fiber, all of which have been confirmed by the operators. As an example, we found that multiple data centers in the same city shared the same fiber bundles, which presents a risk of correlated failures. These risks can be prevented by adding redundant fiber bundles across different cities.

7 Related Work

Structural reliability auditing. The most relevant to CloudCanary are structural reliability auditing systems, INDaaS [70], reCloud [22], and RepAudit [71], which can construct fault graphs from dependency data and perform audits to prevent correlated failures. INDaaS and CloudCanary have higher accuracy than RepAudit and reCloud, because the latter two use approximate algorithms to trade accuracy for efficiency. Moreover, different from all existing work, CloudCanary is designed to perform *incremental auditing* over service updates.

Network/System verification. Failure prevention can also be achieved by formal analysis, such as configuration verification [16, 19, 31, 32, 50, 55, 60, 68], and synthesis/repair [29, 30, 48, 52, 65]. Some of these systems also use incremental verification for speedup when performing analysis [40, 43, 44]. Compared to these systems, CloudCanary has a very different goal—it aims at preventing correlated failures resulting from common dependencies—and also involves completely different algorithms as a result. On the contrary, network verification and synthesis systems primarily focus on reachability and performance properties, such as host-to-host connectivity. Similarly, software misconfiguration detection tools like PCheck [67] also focused on configuration logic, rather than failures caused by common dependencies.

Post-failure diagnostics. Many diagnostic systems [14, 15, 21, 23, 26, 28, 41, 42, 45, 56, 58] and provenance systems [66, 76, 77] have been proposed for failure troubleshooting. CloudCanary aims at a different goal from these efforts.

8 Conclusion

We have presented CloudCanary, a system that can perform real-time audits to prevent correlated failures in service updates. Our system can compute the risk groups in a service snapshot using cached results from previous audits, and it can generate improvement plans with increased reliability. It achieves this using a set of novel techniques, such as incremental auditing and network pruning. CloudCanary outperforms state-of-the-art systems by 200× and can generate improvement plans for large deployments within several minutes. Moreover, it can yield valuable insights over real-world traces from production environments.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Meg Walraed-Sullivan, for their insightful comments. We also thank Nikolaj Bjørner and Bryan Ford for their valuable feedback on the earlier version of this work. This work was partially supported by the National Science Foundation under Grant Numbers CCF-1715387, CCF-1553168, CCF-1302327, and CNS-1801884.

References

[1] Active Geo-Replication. https://docs.microsoft.com/en-us/azure/sql-database/sql-database-active-geo-replication.

[2] ApproxMC. http://www.cs.rice.edu/CS/Verification/Projects/ApproxMC/.

[3] Correlated failures within EBS and EC2. https://aws.amazon.com/message/680342/.

[4] Cost of Data Center Outages. http://datacenterfrontier.com/white-paper/cost-data-center-outages/.

[5] Google: Gmail incident report. https://goo.gl/KY8mjp.

[6] Maxino: A fast MinCostSAT solver. https://github.com/alviano/aspino.

[7] Pyeda. https://github.com/cjdrake/pyeda.

[8] Rack Awareness. https://www.aerospike.com/docs/architecture/rack-aware.html.

[9] Rackspace Outage Nov 12th. https://goo.gl/J98iFz.

[10] Use Canary Tests to Test in Production. https://www.infoq.com/news/2013/03/canary-release-improve-quality.

[11] Walks in the neighborhood: Correlated failure in distributed systems. http://psteitz.blogspot.com/2011/10/correlated-failure-in-distributed.html.

[12] What I wish systems researchers would work on. http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

[13] ZooKeeper Issue Tracker. https://https://issues.apache.org/jira/projects/ZOOKEEPER.

[14] Bahl, P., Chandra, R., Greenberg, A. G., Kandula, S., Maltz, D. A., and Zhang, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2007).

[15] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[16] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2017).

[17] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. Control plane compression. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2018).

[18] Bodík, P., Menache, I., Chowdhury, M., Mani, P., Maltz, D. A., and Stoica, I. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[19] Canini, M., Jovanovic, V., Venzano, D., Novakovic, D., and Kostic, D. Online testing of federated and heterogeneous distributed systems. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2011).

[20] Chakraborty, S., Meel, K. S., and Vardi, M. Y. A scalable approximate model counter. In *19th International Conference on Principles and Practice of Constraint Programming (CP)* (Sept. 2013).

[21] Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-based failure and evolution management. In *1st USENIX Symposium on Networked System Design and Implementation (NSDI)* (Mar. 2004).

[22] Chen, R., Akkus, I. E., Viswanath, B., Rimac, I., and Hilt, V. Towards reliable application deployment in the cloud. In *13th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)* (Dec. 2017).

[23] Chen, X., Zhang, M., Mao, Z. M., and Bahl, P. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[24] Chothia, Z., Liagouris, J., Dimitrova, D., and Roscoe, T. Online reconstruction of structural information from datacenter logs. In *12th European Conference on Computer Systems (EuroSys)* (Apr. 2017).

[25] Cidon, A., Rumble, S., Stutsman, R., Katti, S., Ousterhout, J., and Rosenblum, M. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX Annual Technical Conference (ATC)* (June 2013).

[26] Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[27] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Mar. 2008).

[28] Dunagan, J., Harvey, N. J. A., Jones, M. B., Kostic, D., Theimer, M., and Wolman, A. FUSE: Lightweight guaranteed distributed failure notification. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[29] El-Hassany, A., Tsankov, P., Vanbever, L., and Vechev, M. T. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification (CAV)* (July 2017).

[30] El-Hassany, A., Tsankov, P., Vanbever, L., and Vechev, M. T. NetComplete: Practical network-wide configuration synthesis with autocmpleteion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2018).

[31] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[32] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (May 2015).

[33] FORD, B. Icebergs in the clouds: the *other* risks of cloud computing. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (June 2012).

[34] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010).

[35] FU, Z., AND MALIK, S. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *International Conference on Computer-Aided Design (ICCAD)* (Nov. 2006).

[36] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2011).

[37] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2016).

[38] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *5th ACM Symposium on Cloud Computing (SoCC)* (Nov. 2014).

[39] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing? Lessons from hundreds of service outages. In *7th ACM Symposium on Cloud Computing (SoCC)* (Oct. 2016).

[40] HORN, A., KHERADMAND, A., AND PRASAD, M. Delta-net: Real-time network verification using atoms. In *Proc. NSDI* (2017).

[41] KANDULA, S., KATABI, D., AND VASSEUR, J.-P. Shrink: A tool for failure diagnosis in IP networks. In *MineNet* (Aug. 2005).

[42] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2009).

[43] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *Proc. NSDI* (2013).

[44] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Proc. NSDI* (2013).

[45] KOMPELLA, R. R., YATES, J., GREENBERG, A. G., AND SNOEREN, A. C. IP fault localization via risk modeling. In *2nd USENIX Symposium on Networked System Design and Implementation (NSDI)* (May 2005).

[46] KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27*, 1 (1985), 97–109.

[47] LENERS, J. B., WU, H., HUNG, W., AGUILERA, M. K., AND WALFISH, M. Detecting failures in distributed systems with the Falcon spy network. In *23rd ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2011).

[48] LIU, H. H., WU, X., ZHANG, M., YUAN, L., WATTENHOFER, R., AND MALTZ, D. A. zUpdate: updating data center networks with zero loss. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2013).

[49] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. CrystalNet: Faithfully emulating large production networks. In *26th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017).

[50] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI)* (May 2015).

[51] MAJDARA, A., AND WAKABAYASHI, T. Component-based modeling of systems for automated fault tree generation. *Reliability Engineering & System Safety 94*, 6 (2009), 1076–1086.

[52] MCCLURG, J., HOJJAT, H., CERNÝ, P., AND FOSTER, N. Efficient synthesis of network updates. In *36th ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[53] MERKLE, R. C. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy (IEEE S&P)* (Apr. 1980).

[54] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *3rd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)* (May 2006).

[55] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Mar. 2017).

[56] PEDDYCORD III, B., NING, P., AND JAJODIA, S. On the accurate identification of network service dependencies in distributed systems. In *26th Large Installation System Administration Conference (LISA)* (Dec. 2012).

[57] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[58] REYNOLDS, P., KILLIAN, C. E., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *3rd Symposium on Networked Systems Design and Implementation (NSDI)* (May 2006).

[59] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2015).

[60] TIAN, B., ZHANG, X., ZHAI, E., LIU, H. H., YE, Q., WANG, C., WU, X., JI, Z., SANG, Y., ZHANG, M., YU, D., TIAN, C., ZHENG, H., AND ZHAO, B. Y. Safely and automatically updating in-network ACL configurations with intent language. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2019).

[61] VEERARAGHAVAN, K., MEZA, J., MICHELSON, S., PANNEERSELVAM, S., GYORI, A., CHOU, D., MARGULIS, S., OBENSHAIN, D., PADMANABHA, S., SHAH, A., SONG, Y. J., AND XU, T. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2018).

[62] VELEV, M. N. Efficient translation of boolean formulas to CNF in formal verification of microprocessors. In *Asia South Pacific Design Automation (ASP-DAC)* (Jan. 2004).

[63] VESELY, W. E., GOLDBERG, F. F., ROBERTS, N. H., AND HAASL, D. F. *Fault Tree Handbook*. U.S. Nuclear Regulatory Commission, Jan. 1981.

[64] WU, X., TURNER, D., CHEN, C.-C., MALTZ, D. A., YANG, X., YUAN, L., AND ZHANG, M. NetPilot: Automating datacenter network failure mitigation. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2012).

[65] WU, Y., CHEN, A., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Automated bug removal for software-defined networks. In *14th USENIX Symposium on Networked System Design and Implementation (NSDI)* (Mar. 2017).

[66] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM (SIGCOMM)* (Aug. 2014).

[67] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[68] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *6th USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)* (Apr. 2009).

[69] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error diagnosis by connecting clues from run-time logs. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Mar. 2010).

[70] ZHAI, E., CHEN, R., WOLINSKY, D. I., AND FORD, B. Heading off correlated failures through Independence-as-a-service. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014).

[71] ZHAI, E., PISKAC, R., GU, R., LAO, X., AND WANG, X. An auditing language for preventing correlated failures in the cloud. In *32th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Oct. 2017).

[72] ZHAI, E., WOLINSKY, D. I., XIAO, H., LIU, H., SU, X., AND FORD, B. Auditing the Structural Reliability of the Clouds. Tech. Rep. YALEU/DCS/TR-1479, Department of Computer Science, Yale University, 2013. Available at `http://cpsc.yale.edu/sites/default/files/files/tr1479.pdf`.

[73] ZHAO, X., RODRIGUES, K., LUO, Y., STUMM, M., YUAN, D., AND ZHOU, Y. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *26th ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017).

[74] ZHAO, X., RODRIGUES, K., LUO, Y., YUAN, D., AND STUMM, M. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[75] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2014).

[76] ZHOU, W., FEI, Q., NARAYAN, A., HAEBERLEN, A., LOO, B. T., AND SHERR, M. Secure network provenance. In *23rd ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2011).

[77] ZHOU, W., FEI, Q., SUN, S., TAO, T., HAEBERLEN, A., IVES, Z. G., LOO, B. T., AND SHERR, M. NetTrails: a declarative platform for maintaining and querying provenance in distributed systems. In *ACM International Conference on Management of Data (SIGMOD)* (June 2011).