Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure

Qiao Zhang[1], Guo Yu[2], Chuanxiong Guo[3], Yingnong Dang[4], Nick Swanson[4], Xinsheng Yang[4], Randolph Yao[4], Murali Chintalapati[4], Arvind Krishnamurthy[1], and Thomas Anderson[1]

[1]University of Washington, [2]Cornell University, [3]Toutiao (Bytedance), [4]Microsoft

Abstract

In Infrastructure as a Service (IaaS), virtual machines (VMs) use virtual hard disks (VHDs) provided by a remote storage service via the network. Due to separation of VMs and their VHDs, a new type of failure, called VHD failure, which may be caused by various components in the IaaS stack, becomes the dominating factor that reduces VM availability. The current state-of-the-art approaches fall short in localizing VHD failures because they only look at individual components.

In this paper, we designed and implemented a system called Deepview for VHD failure localization. Deepview composes a global picture of the system by connecting all the components together, using individual VHD failure events. It then uses a novel algorithm which integrates Lasso regression and hypothesis testing for accurate and timely failure localization.

We have deployed Deepview at Microsoft Azure, one of the largest IaaS providers. Deepview reduced the number of unclassified VHD failure events from tens of thousands to several hundreds. It unveiled new patterns including unplanned top-of-rack switch (ToR) reboots and storage gray failures. Deepview reduced the time-to-detection for incidents to under 10 minutes. Deepview further helped us quantify the implications of some key architectural decisions for the first time, including ToR switches as a single-point-of-failure and the compute-storage separation.

1 Introduction

Infrastructure-as-a-Service (IaaS) is one of the largest cloud services today. Customers rent virtual machines (VMs) hosted in large-scale datacenters, instead of managing their own physical servers. VMs are hosted in compute clusters, and mount OS and data VHDs (virtual hard disks) from remote storage clusters via datacenter networks. Resources can be scaled up and down elastically since compute and storage are separated by design.

Achieving high availability is arguably the most important goal for IaaS. Recently, large-scale system design [18, 33, 27], failure detection and mitigation techniques [23, 43, 25, 7, 6, 29, 36], and better engineering practices [10] have been applied to improve cloud system availability. Yet, attaining the gold standard of five-nines (99.999%) VM availability remains a challenge [32, 12].

At Microsoft Azure, there are on the order of thousands of VM down events daily. The biggest category of down events (52%) is what we call VHD failures. Due to compute-storage separation, when a VM cannot access its remote VHDs, the hypervisor has to crash the VM, resulting in a VHD failure. Those VHD failures are caused by various failures in the IaaS stack and constitute the biggest obstacle towards attaining five-nines availability for our IaaS [1].

Compute-storage separation brings unique challenges to locating VHD failures. First, it is hard to find the failing component in a timely fashion, among a large number of interconnected components across compute, storage, and network. The current practice of monitoring individual components is not sufficient. The complex dependencies and interactions among components in our IaaS mean that a single root cause can have multiple symptoms at different places. A network or storage failure may ripple through many other components and affect many VMs and applications. It becomes hard to distinguish causes from effects, resulting in a lengthy troubleshooting process as the incidents get ping-ponged among different teams.

Second, many component failures in the IaaS stack are gray in nature and hard to detect [27]. For failures such as intermittent packet drops and storage performance degradation, some VHD requests that pass through the component can fail but not others. The failure signals in these cases are weak and sporadic in time and space, making fast and accurate detection difficult.

[1] Azure has 34 regions and attains 99.9979% uptime in 2016. [41]

To address these challenges, we have designed and deployed a system called Deepview. Deepview takes a global view: it gathers VHD failure events as well as the VHD paths between the VMs and their storage as inputs, and constructs a model that connects the compute, storage, and network components together. We further introduce an algorithm which integrates Lasso regression [39] and hypothesis testing [14] for principled and accurate failure localization.

We implement the Deepview system for near-real-time VHD failure localization on top of a high-performance log analytics engine. To meet the near-real-time requirement, we add streaming support to the engine. Our implementation can run the Deepview failure localization algorithm in seconds, at the scale of thousands of compute and storage clusters, tens of thousands of network switches, and millions of servers and VMs.

Now in deployment at Azure, Deepview helped us identify many new VHD failure root causes which were previously unknown such as gray storage cluster failure and unplanned Top of Rack switch (ToR) reboot. With Deepview, unclassified VHD failure events dropped from several thousands per day to less than 500, and the Time to Detection (TTD) for incidents was reduced from tens of minutes and sometimes hours to under 10 minutes.

Contributions. We identified VHD failures as the biggest obstacle to five-nines VM availability for our IaaS cloud, and proposed a system to quickly detect and localize them. In particular, we

- Introduce a global-view-based algorithm that accurately localizes VHD failures, even for gray failures.
- Build and deploy a near-real-time system that localizes VHD failures in a timely manner.
- Quantify the implications of key IaaS architectural design decisions, including ToR as a single-point-of-failure and compute-storage separation (Section 7).

2 Background and Motivation

In this section, we first provide background on Azure's IaaS architecture. We explain how compute-storage separation can result in a new type of failure—VHD failures. Then, we introduce the state-of-the-art industry practice in localizing VHD failures and explain how it is slow and inaccurate. Finally, we motivate the approach Deepview takes and explain the challenges for putting the system into production uses.

2.1 Compute-Storage Separation in IaaS

Figure 1 shows Azure's IaaS architecture. A similar architecture seems to be used at Amazon for instances backed by the elastic block store [1]. Every VM has one OS VHD and one or more data VHDs attached. One key design decision is to separate compute and storage physically—**VMs and their VHDs are provisioned from different physical clusters**.

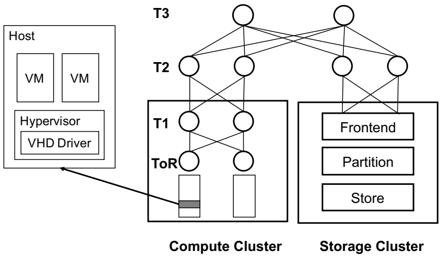


Figure 1: Azure's IaaS architecture. A region has tens to hundreds of compute/storage clusters. Each Tier2 (T2) switch connects some subset of clusters, while Tier3 (T3) switches connect the T2 switches. T3 switches are connected by inter-region network (not drawn).

The main benefit of this separation is to keep customer data available when their VMs become unavailable, e.g., due to a localized power failure. As a result, VM migration becomes easy as we only need to create a new VM (possibly on a different host or cluster) and attach the same VHDs.

In our datacenters, VHDs are provisioned and served from a highly available, distributed storage service [13, 21]. Azure's storage service is deployed in self-contained units of clusters with their own Clos-like network [5, 24, 13], software load balancers, frontend machines and disk/SSD-equipped servers. Similarly, VMs are hosted on physical servers grouped in what we call compute clusters. Each metro region typically has tens to hundreds of compute clusters and storage clusters, interconnected by a datacenter network.

Another benefit is load-balancing. A VM in a compute cluster can remotely mount VHDs from many different storage clusters. A compute cluster therefore uses VHDs from multiple storage clusters, and a storage cluster can serve many VMs from different compute clusters. As we will see later in section 3, this many-to-many relationship is leveraged by Deepview.

VHD Access is Remote. Compute-storage separation requires all VMs to access their VHDs over the network. When a VM accesses its disks, it is unaware that they are remotely mounted. The VHD driver in the host hypervisor provides the needed disk virtualization layer. The driver intercepts VM disk accesses, and turns them into VHD remote procedure call (RPC) requests to the remote storage service. The VHD requests and responses traverse over multiple system components (e.g., the VHD driver and the remote storage stack) and through multiple network hops (e.g., ToR/T1/T2/T3 switches).

VHD Failure	SW Failure	HW Failure	Unknown
52%	41%	6%	1%

Table 1: Breakdown of the causes of VM downtime. VHD failures cause the majority of VM downtime.

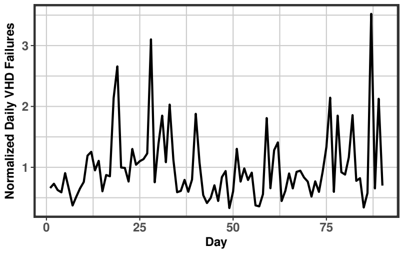


Figure 2: Daily VHD failures normalized the by 3-month average. Every day had at least one failure. On the worst day there were 3.5x more failures than average.

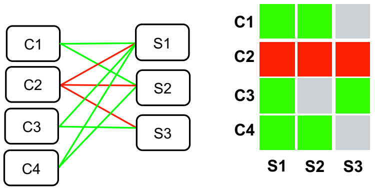
2.2 A New Type of Failure: VHD Failure

Compute-storage separation causes a new type of failure. In our datacenter, whenever VHD accesses are too slow to respond (the default timeout is 2 minutes), the hypervisor crashes the guest OS. In order to protect data integrity, when VHDs do not respond, the guest OS must be paused. But the pause cannot be indefinite—an unresponsive VM can cause customers to fail their own application level SLAs. After some wait, one reasonable option is to surface the underlying VHD access failure to the customer by crashing the guest OS. We call this **VHD failure caused by Compute-Storage Separation** or VHD failure for short.

VHD failure is the biggest cause of unplanned VM downtime. We analyzed an entire year's of IaaS VM down events, including their durations and causes (an internal team finds root causes for VM down events). Table 1 shows that 52% of VM downtime is due to VHD Failures, 41% due to Software Failures (data-plane software and host OS), and 6% due to Hardware Failures, and 1% due to unknown causes.

Figure 2 further shows the daily number of VHD failures normalized by the 3-month average across tens of regions. VHD failures happen daily. Occassionally, they are particularly numerous. The worst day over the 3-month period saw a 3.5x spike in volume.

To minimize the impact of VHD failures and improve VM availability, the most direct approach is to quickly localize and mitigate these failures. Next, we explain the prior VHD failure handling approach and its drawbacks.



(a) Bipartite model (b) Matrix view

Figure 3: The bipartite model and the corresponding matrix view of a downtime event.

2.3 State-of-the-Art: Component View

Our datacenter operators prioritize by the impact of each incident. A large rise in VHD failure events would automatically trigger incident tickets and set off an investigation.

The site reliability engineers (SREs) look at system components individually and locally, to see if any local component anomaly coincides in time with the VHD failure incident. The Compute team might look for missed heartbeats to see if the impacted physical machines have failed. The Storage team might look at performance counters to see if the storage system is experiencing an overload. The Network team might look at network latency and link packet discard rates to determine if some network devices/links could be at fault. Once the failure location is confirmed, the responsible team often has standard procedures for quick mitigation.

Prior to Deepview, failure localization was slow. It was common that we needed tens of minutes, sometimes more than one hour, to localize and mitigate big incidents, and hours to tens of hours to detect and localize gray failures. When a big incident happened, often more than one component had an anomaly because a single root cause could cascade to other services. For example, one big network incident caused as many as 363 related incidents from different services! As a result, the incident ticket could get ping-ponged among the teams.

Further, localization for gray failures [27] was often inaccurate and slow. For example, while we know ToR uplink packet discards can cause VHD requests to fail, it was unclear how severe the discard rate has to be. Setting a threshold to catch those failures became an art: too low generated too many false positives, while too high delayed diagnosis or missed the issue.

3 Our Approach: Global View

Our key insight is that rather than looking at the components individually and locally, we should take a global view. The intuition can be illustrated by the bipartite model in Figure 3a. In this model, we put compute clusters on the left side and storage clusters on the right. We draw an edge from a compute cluster to a storage cluster if it has VMs that mount VHDs from the storage cluster. We also assign an edge weight equal to the fraction of VMs that have experienced VHD failures.

For a compute cluster issue such as an unplanned ToR reboot that causes all VMs under the ToR to crash regardless of what storage clusters they use, we see the edges (highlighted in red) from the impacted compute cluster with high VHD failure rates, as in Figure 3a. When a storage cluster fails, causing all VMs using that storage cluster to experience VHD failures, we see edges with high VHD failure rates coming to the impacted storage cluster.

If we put the compute clusters along the y-axis and the storage clusters along the x-axis, we get a matrix view as shown in Figure 3b. In this matrix view, a horizontal pattern points the incident to the computer cluster, while a vertical pattern points to the storage cluster.

Challenges. Though the bipartite model looks intuitive and promising, there are several challenges to use that insight in a production setting. First, since the bipartite model cannot be easily extended to model the multi-tier network layers, we cannot use it to diagnose failures in the network. Second, while we can use some voting/scoring heuristics to automate the visual pattern recognition, they work well only when the failures are fail-stop. For gray failures [27], fewer VMs would crash so the VHD failure signals are often weaker, and the VHD failure patterns are less clear cut. Third, when big incidents happen, many customers may feel the impact, making timely failure localization imperative. Our system must therefore operate in near-real-time.

Problem Statement. Our goal is to localize VHD failures for both fail-stop and gray failures to component failures in compute, storage or network, at the finest granularity possible (clusters, ToRs and network tiers), all within a TTD target of 15 minutes, in line with our availability objectives.

4 Deepview Algorithm

In this section, we explain how the Deepview algorithm solves the first two challenges—handling network and gray failures. We first describe our new model, a generalization of the bipartite model to include network devices. Then, we introduce our inference algorithm with two main techniques: 1) **Lasso regression** [39] to select a small subset of components as candidates to blame; 2) **hypothesis testing** [14] as a principled and interpretable way to handle strong and weak signals and decide on the components to flag to operators. There are other failure localization algorithms that can be adapted for our problem. We compare Deepview with them in Section 6.2, and show that our approach has better recall and precision.

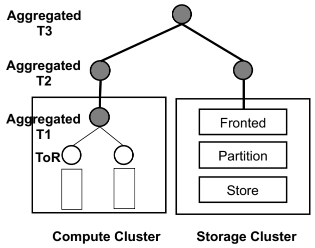


Figure 4: Transforming the Clos network to a tree. Not shown: each aggregated T2 switch connects to many compute/storage clusters and each aggregated T3 switch connects to many aggregated T2 switches.

4.1 Model

In Section 3, we introduced a bipartite model that takes a global view of compute and storage clusters. Here we generalize the model to include network devices.

In this new model, we have three types of components: compute clusters, network devices and storage clusters. Figure 1 shows that compute clusters and storage clusters are interconnected by a number of Tier-2 (T2 for short) and Tier-3 (T3) switches in a Clos topology. ToR and Tier-1 (T1) switches are within the clusters, and are part of the clusters. To model the network, we replace each edge in the bipartite model with a path through the network that connects a compute cluster to a storage cluster. Here we describe the model at the level of clusters (which we call Cluster View in Section 6). We have also extended the model to the granularity of ToRs inside compute clusters (ToR View). For this work, we keep the storage cluster as a blackbox due to its complexity. As future work, we plan to apply our approach to the host level and the storage clusters internals.

4.1.1 Simplify the Clos Network to a Tree

One complication in modeling the network is that each compute/storage cluster pair is connected by many paths. Due to Equal-Cost Multi-path (ECMP) routing [24, 38], we do not know precisely which path a VHD request takes, and therefore, we do not know which path to blame when the request fails.

Our solution is to transform the Clos topology (Figure 1) to a tree topology (Figure 4) so that there is a unique shortest path between each cluster pair. We start from the bottom and go up for each cluster and aggregate the network devices by tiers, and then use shortest path routing to find the lowest overlap between each cluster pair.

The detailed procedure is as follows. First, we start with ToR switches in a cluster and find the T1 switches they connect to. Then, we group those T1 switches as an aggregated T1 group for that cluster. Similarly, we can find the connected T2 switches for those T1 switches and group them as an aggregated T2 group for that cluster. We repeat this procedure to find the aggregated T3 groups. At the end of the aggregation, we have determined the aggregated T1, T2, T3 groups for each cluster in a region. The next step is to find the shortest path for each compute-storage pair . If the aggregated T2 groups of a cluster pair overlap, the midpoint is that overlapped aggregated T2 group; if their aggregated T2 groups do not overlap but their aggregated T3 groups do, the midpoint is the T3 group.

Due to the simplification, we cannot pinpoint to a specific network device, but only to within a network tier. In practice, Deepview is mainly used to decide which SRE teams to notify when VMs crash. Upon notification, network teams have other tools (e.g. Traceroute) to further narrow down to a device for mitigation.

4.1.2 From Paths to Components

Next, we use our observations of VHD failure occurrences to pinpoint which component has failed. We assume that components fail independently, which is a practical and reasonable approximation of the real world. For example, a compute cluster failure is unlikely to be correlated with a storage cluster failure. We can write down a simple probabilistic equation for a path consisting of compute, storage and network components:

$$\mathbb{P}(\text{path i is fine}) = \prod_{j \in \text{path}(i)} \mathbb{P}\left(\text{component j is fine}\right) \quad (1)$$

We approximate $1-\mathbb{P}(\text{path i is fine})$ using the rate of VHD failures observed for that path:

$$\frac{n_i - e_i}{n_i} \approx \prod_{j \in \text{path}(i)} p_j \quad (2)$$

where n_i is the total number of VMs, e_i is the number of VMs that have VHD failures for a given time window, and p_j is the probability that component j is fine. We get a system of equations by writing down (2) for every path. Next, we infer the values of p_j for all components.

We know there is noise in our measurement, so we cannot directly solve the system of equations and would need to explicitly model the noise. Specifically, after taking log on both sides of equation (2) and adding a noise term ε_i, we get a set of linear models:

$$y_i = \sum_{j=1}^{N} \beta_j x_{ij} + \varepsilon_i, \qquad \varepsilon_i \overset{i.i.d.}{\sim} N(0, \sigma^2) \quad (3)$$

where $y_i = \log\left(\frac{n_i - e_i}{n_i}\right)$, $\beta_j = \log p_j$, and the binary variable $x_{ij} = 1$ iff i-th path goes through the j-th component.

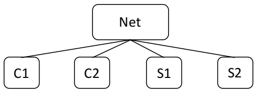


Figure 5: Example where multiple solutions may exist.

Interpretation of β_j: Once we get estimates for β_j, the probability that component j is fine can be computed from β_j because $p_j := \exp(\beta_j)$. If β_j is close to 0, we can clear component j from blame. Otherwise, if β_j is unusually negative, we have strong evidence to blame component j (see Section 4.3). We would ensure $\beta \leq 0$.

Next, we answer the following two questions: (1) how to get fast, accurate, and interpretable estimates for β_j; (2) given the estimates, how to decide which component to blame in a principled and interpretable manner?

4.2 Prefer Simpler Explanation

In practice, the number of unknown variables (β's) can be larger than the number of equations. We illustrate this in a simple example shown in Figure 5. We can list 4 equations with 5 free variables (the βs):

$$\begin{aligned} y_1 &= \beta_{c1} + \beta_{net} + \beta_{s1} + \varepsilon_1 \\ y_2 &= \beta_{c1} + \beta_{net} + \beta_{s2} + \varepsilon_2 \\ y_3 &= \beta_{c2} + \beta_{net} + \beta_{s1} + \varepsilon_3 \\ y_4 &= \beta_{c2} + \beta_{net} + \beta_{s2} + \varepsilon_4. \end{aligned} \quad (4)$$

Suppose all four paths saw equal probability of VHD failures. The blame can be pushed to the compute clusters C1 and C2, or the storage clusters S1 and S2, or the network, or a mix of those. Traditional least-square regression cannot give a solution in this case. But our experience tells us that multiple simultaneous failures are rare for a short window of time (e.g., 1 hour) because individual incidents are rare and failures are (mostly) independent. How do we encode this domain knowledge into our model to help us identify the most likely solution?

To prefer a small number of failures is mathematically equivalent to prefer the estimates $\beta = (\beta_1, \cdots, \beta_N)$ to be sparse (mostly zeros). We express this preference by imposing a constraint on model parameters β. By asking the sum of absolute values of β, i.e., $\|\beta\|_1$ to be small, we can force most of the components of β to zero, leaving only a small number of components of β remaining. This technique of adding a L1-norm constraint is known as Lasso [39], a computationally efficient technique widely used when sparse solutions are needed. We also ensure $\beta \leq 0$ to get valid probabilities. The estimate procedure that encodes all our beliefs in our model is thus the following convex program,

$$\hat{\beta} = \underset{\beta \in \mathbb{R}^N, \beta \leq \mathbf{0}}{\arg\min} \; \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda \, \|\beta\|_1 . \quad (5)$$

Simplicity vs. Goodness-of-fit via λ: This loss function tries to strike a balance between goodness-of-fit in the first term (i.e., how well the model explains the observation) and sparsity in the second term (i.e., fewer failing components are more likely). The regularization parameter λ is the knob. Larger λ prefers fewer components to be blamed at potentially worse goodness-of-fit. The optimal value of λ is set by an automatic (data-adaptive) cross-validation procedure [26].

4.3 Decide Who To Blame

While big incidents are relatively easy to localize with a fixed threshold, it is much harder to find a threshold that can discriminate gray failures from normal components when there are random measurement errors. The estimated failure probabilities for gray failures can be very close to zero (see Section 6.1). The challenge then becomes how big an estimated failure probability is for a gray failure versus just measurement error. Setting such a threshold manually requires laborious data-fitting and is often based on some vague notions of anomaly. In practice, it can be difficult and fragile.

Can we use data to find the decision threshold in a principled and automatic way? Intuitively, the larger the magnitude a (negative) Lasso estimate has, the higher its estimated failure probability, and correspondingly more likely the component has failed. We had a painful experience manually tuning the threshold, but the process gave us some experience in distinguishing true failures (big incidents and gray failures) from measurement noise. We found that if a component's Lasso estimate is much worse than the average, then it is likely a real failure and should be flagged. The further from average, the more confident we are that the component has failed.

This decision can be automated in a hypothesis testing framework. Consider the following one-sided test:

$$H_0(j) : \beta_j = \bar{\beta} \quad \text{v.s.} \quad H_A(j) : \beta_j < \bar{\beta} \tag{6}$$

The null hypothesis $H_0(j)$ says the true probability that component j is fine is no different from the grand average of all components. We then use the data to tell us if we can reject $H_0(j)$ or not. If the data allow us to reject $H_0(j)$ in favor of the alternative hypothesis $H_A(j)$, then we can blame component j. Otherwise, we do not blame component j. The hypothesis test has three steps.

Step 1: Compute Test Statistic. Given Lasso estimates for components in a region, we find the mean $\bar{\hat{\beta}}$ and standard deviation $\sigma_{\hat{\beta}}$. Then we compute a modified Z-score for each component j,

$$z_j = \frac{\hat{\beta}_j - \bar{\hat{\beta}}}{\sigma_{\hat{\beta}}/\sqrt{N}}. \tag{7}$$

Under the assumptions that the measurement error is Gaussian, and other caveats,[2] we approximate the distribution of z_j as a Gaussian distribution with mean zero (under $H_0(j)$) and certain variance.

Step 2: Compute p-value. We then compute the p-value [14] for each component j. The p-value is the probability of seeing a failure probability for component j as extreme as currently observed simply by chance assuming that it is no different from the average. If the p-value is really small, then we do not believe the failure probability for component j is just about average. See the Appendix for more discussion on p-value.

Step 3: Make a Decision. Finally, we apply a standard threshold of 1% on p-value.[3] It expresses our tolerance for false positive rate. For example, if the p-value for component j is less 1%, we blame the component with at most 1% false positive rate. Otherwise, we have insufficient evidence to blame component j.

Avoid the Pitfalls in Multiple Testing. We test every component in a region and flag them based on p-values. For every test, we may falsely blame a normal component with a small chance. But with a large number of components in every region, we are bound to commit an actual false positive if not careful. This is called the multiple testing problem. We use the Benjamini-Hochberg procedure [9] to control the False Discovery Rate. See Appendix for details.

5 Deepview Design and Implementation

We have two main system requirements:

- **Near-real-time (NRT) processing**: VHD failures result in customer VM downtime, so failure localization must be speedy and accurate. We have the requirement that the time-to-detection (TTD) be within 15 minutes.
- **Speedy iteration**: VHD failures are the biggest obstacle to higher VM availability, so there is an immediate need by the operations team for better diagnosis. Our system is designed for quick iteration.

Our system requires two types of input data: non-real-time structural data and real-time event data. The former include the compute and storage clusters information, all the VMs and their VHD storage account information and related context, the paths for all the compute-storage pairs, and the network topology. Taking periodic snapshots of those every few hours suffices for our purposes.

[2] Testing on Lasso estimates is an active research area. We fit a Lasso model to obtain a set of nonzero variables, and refit these variables with least squares. See [46].

[3] Another common threshold is 5%, but it generates too many false positives for testing multiple hypotheses in our setting. See Appendix.

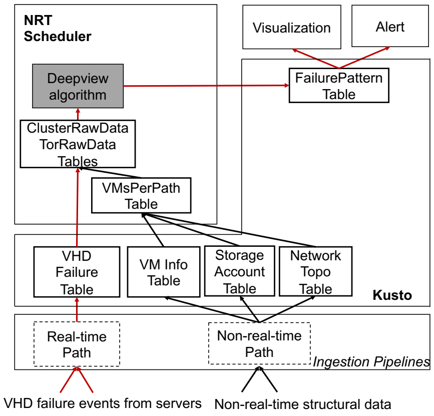


Figure 6: Deepview system architecture. Data schema is given in Table 2.

The latter are the VHD failure signals from servers. To meet near-real-time requirements, our algorithm needs to see VHD failure signals within minutes, ideally through a streaming system.

We need to scale to thousands of compute and storage clusters, tens of thousands of VHD failures per day, tens of thousands of network switches, hundreds of thousands paths, and millions of VMs.

The non-real-time information is either already in our in-house log analytics engine called Azure Kusto [3, 2], or can be generated and ingested into Kusto. Kusto stores data as tables but the tables are append-only, and it supports a SQL-like declarative query language. Kusto is backed by reliable persistent storage from a distributed storage service, using memory and SSD for a read-only cache. By default, it builds indices for all columns to improve query speed.

VHD failure events are generated by hypervisors. They are collected by a real-time pipeline. Since most of our data is already in Kusto, and Kusto provides highly expressive declarative language and fast data analysis, we ingest the VHD failure events into Kusto and build Deepview system on top of it.

System Architecture: The resulting system architecture is shown in Figure 6. It has four components. The real-time path and non-real-time path are for the input data ingestion for the Deepview algorithm. The Kusto platform provides both data analysis and storage for input, intermediate, and output data. The visualization and alert are tools for the consumption of Deepview results. The NRT scheduler is what we build on top of Kusto to support stream processing for the Deepview algorithm.

5.1 Stream Processing

We build our own stream processing system on top of Kusto because most of our data are already there; a few additions to satisfy our needs. We do not claim novelty compared to existing research and commercial streaming systems [44, 8, 4].

Table Name	Schema
VHDFailure	(ts, vm_id, vhd, str_account)
VMInfo	(ts, vm_id, comp_cluster, tor)
StorageAccount	(ts, str_account, str_cluster)
NetworkTopo	(ts, cluster, tor_list, t1_list, t2_list, t3_list)
VMsPerPath	(tstart, tend, num_vms)
ClusterRawData	(tstart, tend, comp_cluster, str_cluster, num_vms, num_failed_vms)
TorRawData	(tstart, tend, comp_cluser, tor, str_cluster, num_vms, num_failed_vms)
FailurePattern	(tstart, tend, region, type, loc, pval, visual_url)

Table 2: Kusto schemas for the Deepview data.

To support stream processing on top of Kusto tables, we use two abstractions:

- A computation directed-acyclic-graph (DAG) declared as a set of SQL-like queries with their output tables.
- A scheduler that runs each query at a given frequency.

We store the DAG and its scheduling policy as tables, since tables are Kusto's only supported data structure.

Computation DAG. The computation DAG consists of a set of queries that read from input tables and produce one or multiple output tables. The queries are the "edges" and the input/output tables are the "nodes". To maintain the DAG in Kusto, we give each query a name and store the query definition and the query output table name in yet another table.

NRT Scheduler. To provide a streaming window abstraction, we use a schedule to describe when each query in the DAG should be executed. The schedule describes how often it should run and how many times to retry. To meet availability requirements, we use a one-hour sliding window that moves forward every 5 minutes.

5.2 Algorithm Implementation

The algorithm implementation has three parts: first, construct the model—instantiate the design matrix x_{ij} and observation y_i based on the Deepview raw data tables, then run Lasso regression to infer β, and finally carry out hypothesis testing to pinpoint the failures.

Sparse Matrix and Region Filtering. The scale of our data poses some challenges for algorithm running time and memory footprint. Constructing a full design matrix requires filling in entries for every path and every component with either zero or one. This can be slow and has high memory usage. However, x_{ij} are mostly zeros since each path has at most tens of components, so we only need to store the non-zero entries. Another simple technique is to only get data from Kusto for regions with non-zero VHD failure occurrences. Since simultaneous failures are rare, region filtering can avoid running the algorithm for some regions without hurting accuracy.

Coordinate Descent. Lasso regression has no closed form solution. Coordinate descent [22] is one of the fastest algorithms to solve the Lasso regression. We minimize the loss function as in Equation 5 with respect to each coordinate β_j while holding all others constant. We cycle among the coordinates until all coefficients stabilize. In practice, with warm start, we found that coordinate descent almost always converges in only a few rounds.

Cross-Validation with Warm Start to Set λ. We set the regularization parameter λ for Lasso using a data-adaptive method, i.e., cross-validation [26]. We use 5-fold cross validation where we split the data by paths into 5 partitions, and use any four of them to fit β for a given choice of λ and then compute the mean squared error (MSE) on the holdout partition using the fitted β. The optimal λ is the one that minimizes the average MSE. We speed up cross validation using a warm start technique [22]. Recall that a larger λ meant fewer non-zero β_j. We start with the smallest λ that turns off all β_j, and then we gradually decrease λ. Since β tends to change only slightly for a small change in λ, we reduce the number of rounds for coordinate descent by reusing $\beta(\lambda_{k-1})$ as the initial values for $\beta(\lambda_k)$.

6 Evaluation

We have deployed Deepview in production at Azure. Here, we first evaluate how well Deepview localizes VHD failures using production case studies. Then, we compare Deepview's accuracy with other algorithms. Next, we analyze various techniques proposed for Deepview and ask how useful each is. Finally, we evaluate how Deepview's runtime efficiency.

6.1 Deepview Case Studies

In this subsection, we ask how effective Deepview is at detecting and diagnosing incidents in production use.

6.1.1 Statistics

We examined the Deepview results for one month. The number of VHD failures generated per day can be up to tens of thousands. For this month, Deepview detected 100 patterns, and reduced the number of unclassified VHD failure events to less than 500 per day. We also tried to associate the detected patterns with incident tickets: 70 of the patterns were directly associated with incident tickets. The other 30 patterns were not associated with tickets. These 30 patterns turned out to be generated by weak VHD failure signals. They were all real underlying component failures that escaped the previous alerting system, either because of their smaller impact (e.g., unplanned ToR reboot) or their gray failure nature (e.g., gray storage failure).

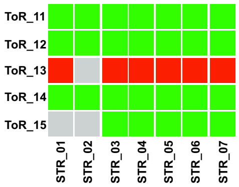


Figure 7: Deepview pattern for an unplanned ToR reboot.

Next we examine some of the representative patterns we found and discuss the insight we learn from them.

6.1.2 Unplanned ToR Reboot

From time to time, ToRs undergo scheduled downtime for firmware upgrade or other maintenance operations. Impacted customers are notified in advance, with their VMs safely migrated to other places. However, occasionally, a ToR may experience an unplanned reboot due to a hardware or software bug. Since each server connects to only one ToR, the VMs under the ToR will not be able to access their VHDs. We get VHD failures as a result. To detect unplanned ToR reboots, Deepview first estimates the failure probability and p-value for the ToR, and then checks the following conditions for confirmation: all the VMs under the ToR get VHD failures, the ToR OS boot time matches the failure time detected by Deepview, and the neighboring ToRs are working fine.

Figure 7 shows one such unplanned ToR reboot detected by Deepview in a small region.[4] It shows a portion of the Deepview UI, which we call ToR view. It clearly shows a horizontal pattern. The ToR switches in the compute cluster are listed on the y-axis and the storage clusters are listed on the x-axis. Each cell in the figure shows the status of the ToR and storage cluster pair. Gray means the VMs under the ToR do not use the corresponding storage cluster; green means the VMs do not have VHD failures; red means the VMs are experiencing VHD failures.

Deepview blamed the right ToR among 288 components in the region (ToRs, T1/T2/T3 switch groups and

[4] Readers may wonder how VHD failure events can be identified when the ToRs are single point of failure. They are in fact stored locally in the servers and are retrieved once network connectivity is restored (typically within 10 minutes for software failures).

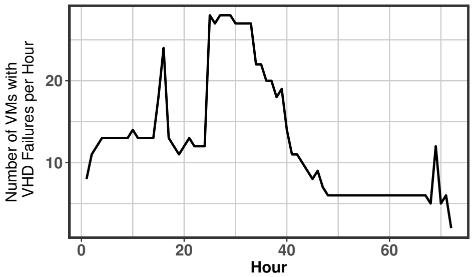


Figure 8: The number of VMs with VHD failures per hour during a storage cluster gray failure.

compute/storage clusters). Deepview estimated the failure probability for the failed ToR to be 100% with a p-value of 1.84E−64, which is much less than 0.01.

Deepview therefore makes it possible to study how often ToRs cause downtime. We discuss this in detail in Section 7.1.

6.1.3 Storage Cluster Gray Failure

Our storage cluster runs a full storage stack including load balancer/frontend, meta-data management, storage layer, etc. VHD failures can happen due to a variety of failure modes in the storage stack. When storage cluster failures are non-fail-stop, the VHD signals can be weak and noisy. For example, the load balancer could discard VHD requests to shed load, and in other cases, software bugs could cause some VHDs to become unavailable, impacting only a subset of VMs.

We next discuss such a storage gray failure case. A new storage cluster was brought online, but with a misconfiguration that allowed a test feature in the caching subsystem to be enabled. This bug mistakenly put some VHDs in negative cache (denoting deletion), rendering them "invisible" and unavailable for VM access.

Based on the VHD failure events at hour 0 in Figure 8, Deepview found three non-zero failure probability entities in the region, 0.34 for storage cluster S0, 0.002 and 0.047 for compute clusters C0 and C1. Notice that because this storage cluster failure only affected a small number of VMs, we did not get a failure probability of 1 for S0. Further, the two compute clusters saw non-zero failure probabilities because they also saw VHD failure events. However, despite the weak signal, our algorithm was able to correctly pinpoint the failure to S0. Our hypothesis testing procedure computed a p-value of 3.9E−34 for S0, identifying it as a failed cluster with very high confidence. On the other hand, C0 and C1 had p-values 0.51 and 0.54, respectively, and signifying a lack of evidence. Using our prior threshold method for detection, we would have delayed the detection by 22 hours. As shown in Figure 8, the signal is weak: the number of VMs affected per hour in the beginning was only around 10, and the peak number was only 28.

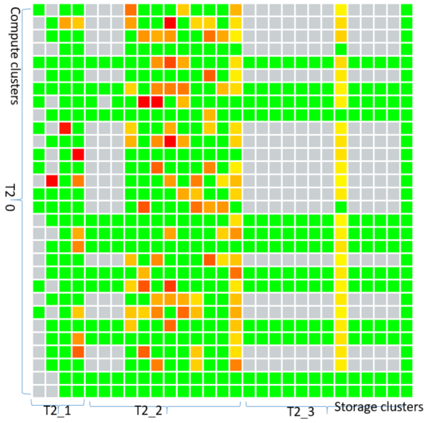


Figure 9: Deepview pattern for a network incident.

6.1.4 Network Failure

In our datacenter, switches other than ToRs have replicas. Single switch failures thus seldom lead to wide impact outages. However, in rare cases, a combination of capacity loss and traffic surge can cause network failures.

In one region, we have over 100 compute clusters and 50 storage clusters. They are connected by four T2 aggregated switches (numbered T2_0 to T2_3) with a T3 aggregated switch (T3_0) on top, as annotated along the axes in Figure 9. Each aggregated switch contains multiple switches. One day, a T3_0 switch underwent a major maintenance event, which triggered some T2 switches in T2_0 to mistakenly detect Frame Check Sequence (FCS) errors on the links to T3_0. Our automatic network service then kicked in and shut down most of links between the T3_0 switch and T2_0 except for three links saved by a built-in safety mechanism.

This loss in capacity together with a surge in storage replication traffic caused significant congestion between T2_0 and T3_0. As a consequence, we saw a significant increase in VHD failures experienced by customer VMs.

Figure 9 shows the pattern in the Deepview UI (partial Cluster View) with compute clusters on the y-axis and storage clusters on the x-axis. The switch aggregated per cluster is annotated on each axis. Yellow cells have a VHD failure rate at most 5%. The VHD failure rates are moderate because the VHD failures in this case were caused by network congestion; most of the time network connectivity was still working.

Deepview identified three aggregated switches with non-zero failure probabilities: 0.21%, 0.11%, 0.03% for T3_0, T2_0, and T2_2, respectively. Their corresponding p-values are 9.91E−12, 4.25E−04, 0.221. We point to T3_0 and T2_0 as the faulty network layers. The failure location is correct, as the root cause is the link congestion between these two network layers. We note Deepview gave small failure probabilities because the VHD failure signals are weak: only a very small percentage of affected VMs crashed. But since T3_0 and T2_0 are high in the network hierarchy, they impact a large number of VMs.

We also experienced network incidents where network connectivity for many VMs were lost. They were easy for Deepview to detect and localize, as the signals were strong: many VMs died at the same time. We present this gray failure case to show the strength of Deepview.

To summarize, we have shown that Deepview can localize various incidents in which the signals can be weak or strong. Deepview has also deepened our understanding of VHD failures by identifying various patterns including horizontal patterns caused by incidents including unplanned ToR reboot, vertical patterns caused by storage outages, and network failure patterns.

6.2 Algorithm Comparison

Several algorithms that have been previously used to localize failures in the network can be extended to localize VHD failures. We compare with two tomography algorithms and a Bayesian network algorithm:

- **Boolean-Tomo [20, 19]:** Classify paths into good and bad paths based on a threshold (bad if at least γ VHD failures). Iteratively find the component on the largest number of unexplained bad paths, as the top suspect until all bad paths are explained. For the threshold γ, we tried $\gamma = 1,2,3,4,5$, and picked $\gamma = 1$ to maximize its recall and then precision.

- **SCORE [31]:** Classify paths into good and bad paths based on a threshold (γ). Iteratively compute for each component its hit ratio $\frac{numBadPaths(c)}{numPaths(c)}$ and coverage ratio $\frac{numUnexplainedBadPaths(c)}{totalNumUnexplainedBadPaths}$. Only consider components above a hit ratio threshold (η). Take the component with the highest coverage ratio as the top suspect. For the threshold γ and η, we tried $\gamma = 1,2,3,4,5$ and $\eta = 0.001, 0.01, 0.1$, and picked $\gamma = 1$ and $\eta = 0.01$ to maximize its recall and then precision.

- **Approximate Bayesian Network [35]:** The runtime to compute exact Bayesian network is exponential in the number of components, and thus is infeasible for us. We tried an approximation [35]. It uses mean-field variational inference to approximate the Bayesian network with a Noisy-OR model, and estimates the component j's failure rate as the posterior mean of a Beta distribution $B(\alpha_j, \beta_j)$. A component is blamed if $\frac{\hat{\alpha}_j}{\hat{\alpha}_j+\hat{\beta}_j}$ is above certain threshold. We do not include its accuracy numbers, because we are unable to make it give meaningful results on our data. The estimated posterior means of component failure rate allows us to apply a threshold. The computation takes 10 minutes for a single region, so this approach is not fast enough for our problem.

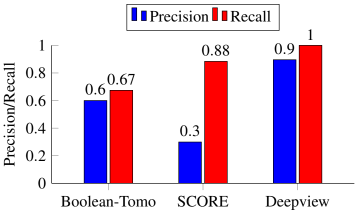


Figure 10: Precision/Recall comparison.

	Compute	Storage	Net	ToR
Precision	0.85	0.875	1.0	1.0
Recall	1.0	1.0	1.0	1.0

Table 3: Precision/Recall by failure type for Deepview.

Dataset. As we cannot run the other algorithms in production, we use trace data to compare algorithms. We had already hand-curated 42 incidents from a detailed study of tickets, so we use trace data from those incidents. They consist of 16 compute cluster issues (not ToR-related), 14 storage cluster issues, 10 unplanned ToR reboots, and 2 network issues. Only time periods when there is an incident are considered because a random sample is too sparse. Thus, we may overestimate the precision. But our comparison is fair since all algorithms use the same baseline ground truth.

Metrics. We compare each algorithm on recall and precision. Recall is the percentage of true failures that have been localized and precision is the percentage of localizations that are correct. In other words, high recall means we can localize most real failures, while high precision means we have few false positives.

Figure 10 summarizes the precision and recall for the 42 incidents. SCORE achieves a recall of 0.88, beating Boolean-Tomo, but it gives many false positives. Deepview, achieves both a high precision of 0.90 and a high recall of 1.0, beating both alternatives. Table 3 shows a breakdown of the precision and recall by failure types for Deepview. Overall, Deepview handles cases with a strong failure signal (Compute/ToR) and those with a weak failure signal (Storage/Network) well. Deepview also does well for unplanned ToR reboots and Network incidents. However, there were fewer of these incidents, so the estimates are to be taken with a grain of salt.

The other advantage of Deepview is that its parameters needs no manual tuning. Parameters are set by cross-validation (for λ) or using a standard interpretable criterion (false positive tolerance of 1% for p-value).

Boolean-Tomo and SCORE, instead, need careful tuning of their thresholds. In fact, we find that their precision and recall are sensitive to the thresholds. We picked those that maximize recall (as recall is typically more important than precision in production), while keeping precision as high as possible. We note that Deepview beats the performance of Boolean-Tomo and SCORE for all combinations of thresholds (omitted for lack of space).

6.3 Deepview Algorithm Analysis

We have introduced a set of techniques for our algorithm. Here, we analyze how useful each technique is.

Cross-validation and λ in Lasso Regression. The regularization parameter λ is set by cross-validation for each region. The optimal values found for incidents in Section 6.2 span three orders of magnitude with a minimum of 0.00012 and a maximum of 0.48. In fact, it is well known in statistical literature that choosing a universally optimal λ for all problems is impossible. The theoretical optimal [11] depends on the number of paths, the number of components, the structure of the network, and the error variance (i.e., how stable are VHD failures among different paths). When cross-validation is fast, it is preferred to a manual threshold.

Hypothesis Testing and Gray Failures. We use hypothesis testing to find a decision threshold to localize both big incidents and gray failures in the presence of random noise. The gray failure case studies in Section 6.1 show that hypothesis testing is essential. For the storage case, the failure probabilities are 0.34 for the truly failed storage cluster S0, and 0.002 and 0.047 for two normal compute clusters. Their p-values 3.9E−34 and 0.51 and 0.54 are needed to accentuate the difference and allow us to pick only S0. Similarly, for the network case, looking at p-values allow us to filter out T2_2.

6.4 Deepview Running Time

Algorithm Running Time. We measure the running time for Deepview algorithm in production. The worst-case running time is 18.3 seconds on a single server. It includes the time to read input data from Kusto, execute the algorithm and write the output data to Kusto.

Time to Detection (TTD) TTD is defined as the time between when an incident happens and when the failure is localized. The average time from a VHD failure event to its appearance in Kusto is 3.5 minutes. Adding the 5 minutes windowing time and the processing time, Kusto achieves a TTD under 10 minutes. This is a significant improvement over the previous TTD which typically lasted from tens of minutes to hours.

7 Discussion

Several architectural decisions were made when our IaaS was built. One is that a server connects to only a single ToR via a single NIC. While this makes ToR a single-point-of-failure (SPOF), the decision dramatically reduces networking cost. Another decision is that a VM can host its VHDs in any storage cluster in the same region. This makes load-balancing for storage clusters easy, but with potentially higher network latency and lower throughput. Further, both decisions may adversely impact VM availability. Using the data collected from Deepview, we can now study the impact of these decisions quantitatively.

7.1 ToR as a Single-Point-of-Failure

As we have described in Section 6.1, Deepview can detect unplanned ToR reboots. From the failure patterns, we find that there are two types of ToR failures: soft failures and hard failures. Soft failures can be recovered by rebooting the ToR, while hard failures cannot.

Our data shows that: (1) less than 0.1% switches experience unplanned reboots in a month; (2) 90% of the failures are soft failures, with the rest hard failures. The hard failure rate agrees with our ToR Return Merchandise Authorization (RMA) rate, which indicates that 0.1% switches need to be RMAed in one year. These numbers are obtained from a fleet of tens of thousands of ToRs.

The impact of a soft failure typically lasts for less than 20 minutes: 10 minutes for the ToRs to come up and 10 minutes for the VMs to recover. The impact of a hard failure lasts longer as the failed switch needs to be physically replaced. The impact to VMs can be shorter though as the VMs can be migrated to other hosts due to the separation of compute and storage. We conservatively use 2 hours as the impact period for hard failures.

If the ToR is the only failure source for VMs on that rack, the availability of our IaaS is no better than

$$1 - \frac{0.9 \times 20 + 0.1 \times 120}{1000 \times 30 \times 24 \times 60} = 99.99993\%$$

Even with ToR as the single point of failure, the service can achieve six-nines. This meets the rule of thumb that critical dependencies need to offer one additional 9 relative to the target service [40].

Thanks to Deepview data, for the first time, we are able to show that ToR as a single point of failure is an acceptable design choice for IaaS as it is not on the critical path for five-nines availability.

Note that simply examining ToR logs would not have given us these numbers, as many ToR reboots are planned, with no impact on VM availability.

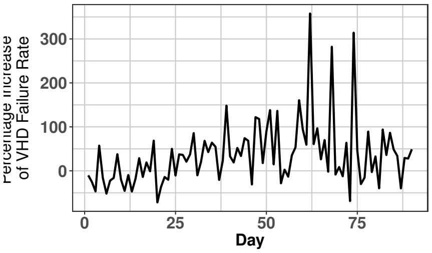


Figure 11: Daily percentage increase in VHD failure rate for VMs crossing T3 and above compared to those that only cross T2 for a 3-month period.

7.2 Co-locate or Disaggregate?

A VM can use VHDs from any storage cluster in the same region, due to the separation of compute and storage. We look at the network distance between VMs and the storage clusters for their VHDs. We find that some 51.8% of VHD paths go through T2, 41.0% need to go through T3 and the rest go above T3 in Azure.

A longer network path may result in higher network latency and packet drop rate. However, it is not clear whether it will also negatively affect VM availability.

Here we use the Deepview data to answer this quantitatively. We look at our data for three months. For each day, we first compute the VHD failure rates r_0 and r_1 for VMs crossing T2 only and VMs crossing T3 and above, respectively. Then, we find the percentage increase $(r_1 - r_0)/r_0$.

Figure 11 shows the daily percentage increase over a 3-month period. VMs whose network paths cross T3 network layer or above see a higher VHD failure rate than those that only need to cross T2 on most days. There is a 11.4% increase ($(\overline{r_1} - \overline{r_0})/\overline{r_0}$) in the VHD failure rate if the VHD access needs to cross T3 or above.

One possible explanation is that as VHD requests go up the network tiers, they traverse more switches which may become oversubscribed. Thus VHD requests may become more likely to fail when network path lengths get longer. An implication of this study is that there is some benefit to colocating VMs and their VHDs in nearby clusters for availability.

8 Related Work

Machine Learning. Machine learning techniques have been used for failure localization, such as decision trees [6, 17], Naive Bayes [45], SVM [42], correlations [43], clustering [16], and outlier detection [36]. They allow domain knowledge to be encoded as features, but in general require a rich set of signals to discriminate different failure cases and may rely on assumptions about traffic that are not generally applicable. The most relevant work is NetPoirot [6], which targets a similar scenario as ours, but with a very different approach. NetPoirot is a single node solution where end-hosts independently run pre-trained classification models on local TCP statistics to infer failure locations. We believe NetPoirot and Deepview are complementary—TCP metrics from IaaS VMs may provide a useful signal to Deepview.

Tomography. There has been a large body work in network tomography (see [15] for a survey), and specifically binary tomography and its variants [20, 19, 31] for network failure localization. Typically, greedy heuristics are used to select among multiple solutions that all explain the observations. Various thresholds are often needed to tradeoff between precision and recall ratios. Compared with those approaches, Deepview avoids manual threshold tuning and achieves both higher recall and precision as shown in section 6.2.

Bayesian Network. Bayesian network [34] is a principled probabilistic approach to failure localization. It can model complex system behaviors [7] and handle measurement errors [28]. While exact inference is intractable [30], there are various approximation techniques such as using noisy-or to simplify conditional probability calculation [35, 7, 37], considering *k*-subset root-causes to shortcut marginalization [28, 7], using a simple factored form for joint posterior [35], or using message passing for faster inference [37]. For our problem, we find that using a combination of approximation techniques (we tried two [35]) was essential. It is future work to compare Deepview with some practical Bayesian network approach.

9 Conclusion

We identified VHD failures caused by compute-storage-separation as the main factor that reduces VM availability at our IaaS cloud. We introduced Deepview, a system that quickly localizes failures from a global view of different system components and a novel algorithm integrating Lasso regression and hypothesis testing. Data from production allowed us to quantitatively evaluate precision and recall across many failure events. We also used Deepview data to evaluate the impact of system architecture on VM availability.

Acknowledgement

We thank our Azure colleagues Brent Jensen, Girish Bablani, Dongming Bi, Rituparna Paul, Abhishek Mishra, Dong Xiang for their valuable discussions and support. We thank our MSR colleagues Pu Zhang, Myeongjae Jeon and Lidong Zhou, and intern Jin Ze for their contributions to an early prototype of Deepview. We thank our shepherd Mike Freedman and the anonymous reviewers for their feedback. This work was partially supported by the NSF (CNS-1616774).

References

[1] Amazon EC2 Root Device Volume. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts.

[2] Azure Kusto (Preview). https://docs.microsoft.com/en-us/connectors/kusto/.

[3] Introducing Application Insights Analytics. https://blogs.msdn.microsoft.com/bharry/2016/03/28/introducing-application-analytics/.

[4] ABADI, D. J., CARNEY, D., ETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. Aurora: a New Model and Architecture for Data Stream Management. *The VLDB Journal 12* (2003), 120–139.

[5] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM* (2008).

[6] ARZANI, B., CIRACI, S., LOO, B. T., SCHUSTER, A., AND OUTHRED, G. Taking the Blame Game Out of Data Centers Operations with NetPoirot. In *SIGCOMM* (2016).

[7] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM* (2007).

[8] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD Conference* (2005).

[9] BENJAMINI, Y., AND HOCHBERG, Y. Controlling the False Discovery Rate: a Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* (1995), 289–300.

[10] BEYER, B., JONES, C., PETOFF, J., AND MURPHY, N. R. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[11] BICKEL, P. J., RITOV, Y., AND TSYBAKOV, A. B. Simultaneous Analysis of Lasso and Dantzig Selector. *The Annals of Statistics* (2009), 1705–1732.

[12] BISHOP, T. Microsoft Says Google's Cloud Reliability Claim vs. Azure and Amazon Web Services Does Not Compute, 2017. https://www.geekwire.com/2017/microsoft-says-googles-cloud-reliability-claim-vs-azure-amazon-web-services-not-compute.

[13] CALDER, B., ET AL. Windows Azure Storage: a Highly Available Cloud Storage Service with Strong Consistency. In *SOSP* (2011).

[14] CASELLA, G., AND BERGER, R. L. *Statistical Inference*, vol. 2. Duxbury Pacific Grove, CA, 2002.

[15] CASTRO, R., COATES, M., LIANG, G., NOWAK, R., AND YU, B. Network Tomography: Recent Developments.

[16] CHEN, M. Y. ., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN* (2002).

[17] CHEN, M. Y., ZHENG, A. X., LLOYD, J., JORDAN, M. I., AND BREWER, E. A. Failure Diagnosis Using Decision Trees. In *ICAC* (2004).

[18] DEAN, J. Designs, Lessons and Advice From Building Large Distributed Systems. *Keynote from LADIS 1* (2009).

[19] DHAMDHERE, A., TEIXEIRA, R., DOVROLIS, C., AND DIOT, C. NetDiagnoser: Troubleshooting Network Unreachabilities Using End-to-end Probes and Routing Data. In *CoNEXT* (2007).

[20] DUFFIELD, N. Network Tomography of Binary Network Performance Characteristics. *IEEE Transactions on Information Theory 52*.

[21] 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 *OSDI* (2010).

[22] FRIEDMAN, J., HASTIE, T., AND TIBSHIRANI, R. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software 33*, 1 (2010), 1.

[23] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or Die: High-Availability Design Principles Drawn From Googles Network Infrastructure. In *SIGCOMM* (2016).

[24] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[25] GUO, C., ET AL. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM* (2015).

[26] HASTIE, T. J., TIBSHIRANI, R., AND FRIEDMAN, J. H. The elements of statistical learning: data mining, inference, and prediction, 2nd Edition. In *Springer series in statistics* (2009).

[27] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *HotOS* (2017).

[28] KANDULA, S., KATABI, D., AND VASSEUR, J.-P. Shrink: a Tool for Failure Diagnosis in IP Networks. In *MineNet* (2005).

[29] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed Diagnosis in Enterprise Networks. In *SIGCOMM* (2009).

[30] KOLLER, D., AND FRIEDMAN, N. Probabilistic Graphical Models - Principles and Techniques.

[31] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. IP Fault Localization via Risk Modeling. In *NSDI* (2005).

[32] LEOPOLD, G. AWS Rates Highest on Cloud Reliability, 2015. https://www.enterprisetech.com/2015/01/06/aws-rates-highest-cloud-reliability.

[33] MOGUL, J. C., ISAACS, R., AND WELCH, B. Thinking About Availability in Large Service Infrastructures. In *HotOS* (2017).

[34] PEARL, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 1988.

[35] PLATT, J. C., KICIMAN, E., AND MALTZ, D. A. Fast Variational Inference for Large-scale Internet Diagnosis. In *NIPS* (2007).

[36] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive Realtime Datacenter Fault Detection and Localization. In *NSDI* (2017).

[37] STEINDER, M., AND SETHI, A. S. End-to-end Service Failure Diagnosis using Belief Networks. In *NOMS* (2002).

[38] THALER, D., AND HOPPS, C. Multipath Issues in Unicast and Multicast Next-Hop Selection, 2000. IETF RFC 2991.

[39] TIBSHIRANI, R. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.

[40] TREYNOR, B., DAHLIN, M., RAU, V., AND BEYER, B. The Calculus of Service Availability. *ACM Queue 15* (2017).

[41] VISWAV, P. Microsoft Dismisses Google's Cloud Reliability Claim, 2017. `https://mspoweruser.com/microsoft-dismisses-googles-cloud-reliability-claim`.

[42] WIDANAPATHIRANA, C., LI, J. C., SEKERCIOGLU, Y. A., IVANOVICH, M. V., AND FITZPATRICK, P. G. Intelligent Automated Diagnosis of Client Device Bottlenecks in Private Clouds. *2011 Fourth IEEE International Conference on Utility and Cloud Computing* (2011), 261–266.

[43] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI* (2011).

[44] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP* (2013).

[45] ZHANG, S., COHEN, I., GOLDSZMIDT, M., SYMONS, J., AND FOX, A. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *DSN* (2005).

[46] ZHAO, S., SHOJAIE, A., AND WITTEN, D. In Defense of the Indefensible: A Very Naive Approach to High-Dimensional Inference. *arXiv preprint arXiv:1705.05543* (2017).

A P-value Correction for Multiple Testing

To decide if a component has failed, we could make a decision based on a threshold for the estimated failure probability for that component. But we can make a more principled decision by conducting a hypothesis test for each component as specified in (6). In this appendix, we explain the details in doing this testing. We explain our p-value, and motivate and explain how we do multiple testing.

A.1 Interpretation of p-values

To conduct the test for each component, we construct the test statistics as in (7) for each of the N components. We then compute the p-value for each test to decide whether to reject the null hypothesis. The p-value is defined as, the probability, assuming the null hypothesis is true, of the sampling test statistic having a value at least as extreme as observed. If the null hypothesis is true, we should expect a moderate p-value. However, if the computed p-value is small, we have evidence to believe that the null hypothesis is false. In fact, when the p-value is too small (e.g., below the conventional 1%, 5%, and 10% significance level), we should reject the null hypothesis $H_0(j)$, since it is highly unlikely that it can explain the values we have observed.

If the p-value is greater than the significance level, then the test is inconclusive. However, we give extra attention to borderline cases to decrease the false negative rate. For example, we produce warnings with lower priority for those components whose p-values are only slightly greater than the significance level.

A.2 Choice of Significance Level

How to choose an appropriate significance level? For testing a single hypothesis, conventional choices of significance level include 1%, 5%, and 10%.

However, when testing multiple hypotheses, we need to be more careful about false positives. Suppose we are testing 100 null hypotheses, all of which are true. If we use 5% as the significance level, then there is roughly 5% probability that we incorrectly reject the null hypothesis—committing a false positive. Further, if these 100 tests are independent, then we are almost certain to make at least one false positive:

$$\begin{aligned}\mathbb{P}(\text{at least one false positive}) &= 1 - \mathbb{P}(\text{no false positive}) \\ &= 1 - 0.95^{100} = 0.994.\end{aligned} \tag{8}$$

Intuitively, the more hypotheses we test simultaneously, the more likely we are to make a mistake.

To reduce the tendency of making mistakes when testing multiple hypotheses, we need to provide a stricter significance level than a single test. This is called the multiple testing correction.

A.3 Multiple Testing Correction

There are two approaches to multiple testing correction: family-wise error rate (FWER) control correction or false discovery rate (FDR) control correction. We use FDR control in Deepview algorithm since it is the more powerful alternative.

Let V be the number of false positives (the healthy components that we falsely blame), and R be the number of rejected hypotheses (the total number of components we blame). Then the false discovery rate (FDR) is defined as

$$FDR := \mathrm{E}[Q] := \mathrm{E}[V/R] \tag{9}$$

The Benjamini-Hochberg procedure [9] is the most popular FDR control procedure due to its simplicity and effectiveness. The procedure is as follows:

1. Do N individual tests and get their p-values P_1, $P_2, \ldots, P_N$ corresponding to null hypothesis $H_0(1)$, $H_0(2), \ldots, H_0(N)$.
2. Sort these p-values in ascending order and denote them by $P_{(1)}, P_{(2)}, \cdots, P_{(N)}$.
3. For a given threshold on FDR α, find the largest K such that $P_{(K)} \leq \frac{K}{N}\alpha$.
4. Reject all null hypotheses for which their p-values are smaller than or equal to $P_{(K)}$.

This procedure controls the FDR under α.