Predictive and Adaptive Failure Mitigation to Avert Production Cloud VM Interruptions

Sebastien Levy[†], Randolph Yao[†], Youjiang Wu[†], Yingnong Dang[†], Peng Huang[◇]
Zheng Mu[†], Pu Zhao[⋆], Tarun Ramani[†], Naga Govindraju[†], Xukun Li[†]
Qingwei Lin[⋆], Gil Lapid Shafriri[†], Murali Chintalapati[†]

[†]*Microsoft Azure* [◇]*Johns Hopkins University* [⋆]*Microsoft Research*

Abstract

When a failure occurs in production systems, the highest priority is to quickly mitigate it. Despite its importance, failure mitigation is done in a reactive and *ad-hoc* way: taking some fixed actions only after a severe symptom is observed. For cloud systems, such a strategy is inadequate. In this paper, we propose a preventive and adaptive failure mitigation service, NARYA, that is integrated in a production cloud, Microsoft Azure's compute platform. Narya *predicts* imminent host failures based on multi-layer system signals and then decides smart mitigation actions. The goal is to *avert* VM failures. Narya's decision engine takes a novel online experimentation approach to continually explore the best mitigation action. Narya further enhances the adaptive decision capability through reinforcement learning. Narya has been running in production for 15 months. It on average reduces VM interruptions by 26% compared to previous static strategy.

1 Introduction

Failures are common in large systems. High-availability system designs require techniques that address a key question: *once a failure occurs, how to quickly detect and mitigate it so the system can continue running?* Mitigating a failure here means attempting to make the failure symptom disappear without necessarily diagnosing and fixing the underlying bugs first. However, for a large cloud infrastructure like Microsoft Azure that serves millions of customers running virtual machines and various software atop, only employing post-failure detection and mitigation techniques is insufficient.

This is because if a system only takes mitigation actions after a failure is detected, users may already be having bad service experience as the system runs in a degraded mode (not completely failing) [15, 17, 29]. Moreover, when a failure is detected, the system will be under intense pressure to mitigate the failure fast in order to minimize downtime; but in practice failure mitigation takes time for large systems, and expediting mitigation could even worsen the situation [12]. In addition, our experience suggests that even short, mitigated failures can be impactful to customers due to the interruptions themselves.

Therefore, cloud systems should also design techniques to address the question of, *whether a failure may be imminent, and if so, what preventive actions should be taken to* ***avert*** *this failure?* Several recent works tackle the failure prediction problem [14, 27, 38] in the context of disk failures. But they focus on prediction alone, with the goal of alerting operators or providing allocation hints [25]. The questions of how much benefit does the prediction bring, and more importantly what preventive mitigation actions should the system take in response to predicted failures remain open. Answering these questions requires a holistic solution—one that is closely integrated in the system's control loop, which not only predicts host failures in real time, but also automatically decides the proper mitigation actions, measures the benefits, and continuously adjusts its actions based on the measured benefits.

In this paper, we present NARYA to fill this aforementioned gap. Narya is an end-to-end service with predictive and smart failure mitigation fully integrated in the Azure compute platform for its Virtual Machine (VM) host environment. The design goal of Narya is to *prevent* VM failures ahead of time and enhance the self-managing capability of the Azure compute platform for providing smooth VM experience to customers.

Narya's design is informed by several observations we had. First, while failure mitigation is a crucial step in cloud service operation, the current practice is *ad-hoc*. To mitigate a (predicted) failure, developers use static policies that prescribe actions based on the symptoms and domain knowledge. While such approach is effective for simple systems, it does not work well at Azure scale. With multi-tenancy, heterogeneous infrastructure components, and diverse customer workloads, it is difficult to comprehensively categorize different failure scenarios in a large cloud system beforehand and determine good mitigation actions (or their parameters).

Moreover, as the cloud system is constantly changing (software/hardware updates, customer workload changes), some mitigation action that worked well in the past may no longer be optimal. As a result, developers have to reactively keep adjusting the actions based on hind sights from service incidents.

For example, initially restarting a host node upon receiving a predictive failure signal may be effective as the system failures tend to be caused by some transient hardware issues; but gradually permanent node failures become more common so restarting is not the best mitigation action anymore—live migrating the virtual machines from the node predicted to fail to a healthy node may be a better action. Therefore, for cloud-scale systems, we need smart and adaptive failure mitigation.

To realize this goal, our insight is that the effectiveness of taking a particular mitigation action in a complex and changing system is generally probabilistic as there are simply too many factors affecting it (the network condition, virtual machine size, applications, hardware specs, customer activities, etc.). We usually do not know beforehand whether some mitigation action is good or not, or whether there is a better action, unless we try it. Consequently, explorations with *production* workload is indispensable to determine the (near-)optimal failure mitigation action. Nevertheless, we should ensure that over time the actions taken maximize the expected effectiveness (minimize the potential customer impact).

Based on this insight, Narya takes a novel online experimentation approach. In particular, Narya predicts whether host nodes in the production fleet may likely fail and then leverages A/B testing strategy to continually experiment with different mitigation actions, measure the benefits, and discover optimal actions. The rationale behind the A/B testing strategy is that it, in essence, introduces randomization that avoids biased pivots of the diverse nodes, which helps surface the statistically significant effective actions.

One important drawback of the A/B testing strategy is its cost of exploring each action until statistical significance is found and then always choosing the estimated best action. This problem is essentially the classic *exploration–exploitation* trade-off [32] in learning systems that need to make decisions with incomplete information (about the system stack, customer workloads, etc.), constant changes, and uncertain pay-offs (whether the action will prevent future failures). The dilemma is whether the learning agent should repeat a mitigation action that has worked well so far, *i.e.*, *exploit*, or it should try some novel choices in the hope of getting better rewards, *i.e.*, *explore*. We address this issue by enhancing the Narya decision engine using a dynamic assignment learnt through a multi-armed Bandit model [2, 33]. This helps decrease overall cost by better leveraging the early cost estimation of each action and by continuously exploring each of them to adapt to system changes.

Narya has been running in production for 15 months in Azure as part of the Gandalf [24] suite. Narya successful prevents many VM interruptions for customers. In nine production experiments that Narya runs for different failure types, Narya on average reduces VM interruptions by 26% compared to the previous static strategy. This reduction is close to what the optimal strategy could achieve (35%).

The major contributions of this work are:

- We propose a holistic failure avoidance solution that includes failure prediction, new failure mitigation actions, and intelligent mitigation strategies.
- We design a novel approach of using A/B testing for online experimentation with production workload to automatically identify good failure mitigation actions.
- We explore a more advanced reinforcement learning approach to optimize choice of mitigation action.
- We evaluate the proposed solution in a large-scale, production cloud service, Azure, to validate its effectiveness.

2 Background and Motivation

A traditional system's operation cycle is as follows: a failure is detected; developers diagnose the failure and find out the root cause; a patch is written; the system is re-deployed. For cloud systems, operating in this exact sequence is problematic because the time it takes to identify the root cause and develop a fix is usually long and exceeds the downtime budget. Instead, once a failure is detected, some *mitigation* action like restart will be applied first without necessarily knowing the bug.

2.1 Target System and Goal

We tackle the problem of preventive and smart failure mitigation for cloud systems. Our specific target system is the VM host environment, a node, in the Azure compute platform. The host environment is a complex stack consisting of guest OSes, guest agents, hypervisor, host OS, host agents, firmware, and hardware. The node is backed by locally attached disks and remote virtual disks. Each node is connected to various compute services, together referred to as controller, that is responsible for provisioning resources and performing management actions such as creating and destroying VMs.

Azure already employs layers of monitoring mechanisms to actively detect if a host node has failed (e.g., via periodic pings), and mitigate the failure with actions such as rebooting the node. We aim to further develop techniques that *predicts* whether a host environment might fail soon and *automatically* decides an appropriate mitigation plan among multiple choices. The end goal is to *avoid* future VM failure events.

2.2 Are Failures Predictable?

To predict failures, there are two basic requirements: (i) the imminent failure is not abrupt; and (ii) there is telemetry recorded to indicate the degradation. One type of predictable hardware issue is certain hardware parts wear out. We could predict using the age or the wear-out rate. Combined with other system signals such as workload patterns, we can predict if a host will fail soon. Resource leak, including memory/file handle/network ports leak, is a common type of predictable software failure. We could predict them using the resource usage trend. If failures are correlated with certain hidden factors such as timeout settings, bugs related to timers, and release schedule, they may also occur on a predictable basis.

Timestamp	Event
03-14 18:00:00	A node with 16 VMs was predicted to fail with 0.7 prob.
03-16 01:09:12	Node agent crashed, 17 VMs offline
03-16 01:15:26	Controller probes to node agent timed out, retry
03-16 01:31:00	Node state marked by controller as unhealthy
03-16 02:07:37	Controller tried to recover the node through restart
03-16 02:23:02	Failed to receive node reboot success signal
03-16 02:23:33	VMs in the node were recreated in another node
03-16 04:40:17	Node was sent out for repair
03-16 06:13:21	Offline diagnosis finished, disk fault was suspected

Table 1: Events timeline for a production node.

2.3 Why Reacting on Predicted Failure?

Since predicted failure is about something (complete failure) that has not occurred yet, one option is to only treat it as an early warning and not act on it. After all, it is developers' common mindset that *"If It Ain't Broke, Don't Fix It"*. However, for cloud services, this mindset puts customers and their VMs at the risk of suffering interruptions. With techniques such as live migration which can migrate a VM from one node to another with minimal customer impact, cloud vendor is in a better position to help customer avoid failures.

To give an example, Table 1 shows a production case of the timeline for events in a node. In this case, the node was predicted to fail with a relatively high probability, but no preventive action was taken. So both the existing VMs and new VMs still run in this node. Later, this node indeed failed (OS crash), which caused 17 VMs including 1 new VM to be offline. The controller tried to probe the VMs and timed out. Then it tried to reboot the node but failed. Finally, the controller decided to recreate the VMs in another node. Subsequent offline diagnosis confirmed the disk on the node was indeed problematic. Had we taken some mitigation action when receiving the failure prediction signal, we could have saved the long VM disruptions and customer impact.

2.4 Why Static Mitigation Is Insufficient?

Intuitively each predicted failure should be handled in the same way using an optimal method. Indeed, initially we used a static rule where all predicted bad nodes would be mitigated using the same plan: 1) block the allocation on the node; 2) try to live migrate (LM) VMs; 3) wait for 7 days for short-lived VMs to be destroyed by customers; 4) force migration of remaining VMs; 5) mark the node offline and send it for repair. Although this plan looks reasonable, it quickly faces limitations. Blocking allocation results in capacity pressure while for some predicted failures, avoiding allocation may be better. Some failures may be too urgent to use LM (*e.g.*, disks are too bad), or be better mitigated through a quick reboot. Forced migration causes unnecessary customer impact if nodes still healthy after 7 days are false positives. Marking nodes offline is also suboptimal when capacity is low.

Using static assignment prevents us from knowing what would have happened if we chose a different action, and therefore we would not be able to know how much customer pain

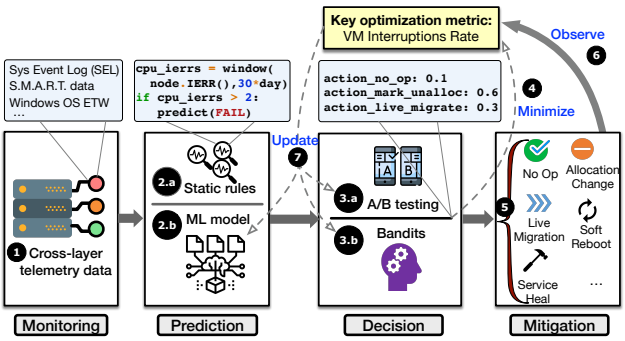


Figure 1: High-level workflow of Narya.

we save or if we are using the best action. In addition, static mitigation suffers from the problem of system changes over time. With the complexity of our cloud system, the effect of a rule, as well as the telemetry that a rule relies on, is constantly changing. In these cases, since static mitigation does not try other actions, it could increase customer pain without us realizing it. For example, a low CPU frequency can be a defense mechanism from a CPU to indicate an imminent failure; detecting such drop in CPU frequency and applying a mitigation action can be beneficial. However, new improvements in the system could voluntarily decrease CPU frequency to conserve energy. In this case, the original rule could have a high number of false positive in prediction results, and applying the same action will very likely cause more harm than good.

Another limitation of static rule-based failure mitigation is that it creates tendency for developers to make *ad-hoc* modifications to the rules based on some isolated cases. No mitigation action can be perfect and customers may complain if they suffer from such mitigation. In such case, developers tend to modify the rule to satisfy the customers. However, without testing if that change does reduce the *overall* customer pain, it is possible that the situation gets worse and the policy might be switched back again when another customer complains about this new policy.

3 Overview

We design Narya, an end-to-end service that is integrated in the Azure compute platform, to predict host failures and automatically decide what mitigation actions to take for the predicted failures. The design goal of Narya is to *avert* potential VM failures while minimizing the impact to customers. Narya advances the current practice of failure mitigation in two ways: (i) replacing existing static and *ad-hoc* mitigation rules to adaptive and systematic decision algorithms; and (ii) transforming the traditionally reactive, post-failure mitigation activity to proactive failure avoidance mechanisms.

3.1 Narya Workflow

Narya takes a novel online experimentation and learning approach to the failure mitigation problem. Figure 1 shows overview of the high-level workflow in Narya.

Each node in Azure is deployed with monitoring agents that collect various telemetry signals about the host environment (❶, §4.1). The prediction component in Narya continuously consumes these signals and predicts whether some node will fail soon (❷). The prediction is made by both static domain rules (2.a, §4.2) and running machine learning inference (2.b, §4.3). Each prediction result is streamed into the decision component in Narya as a mitigation request. Narya supports two decision schemes: A/B testing (3.a, §6.1), and Bandit model (3.b, §6.2). The decision component computes a probability distribution of applicable mitigation action choices (❹, §5) and then picks an action based on the distribution. The mitigation controller applies the chosen mitigation action to the suspected node (❺). This whole process is an automated feedback loop that optimizes key a objective metric—VM interruption rate (§3.2). Narya observes (❻) the effect of the mitigation actions and adapts (❼) future prediction and mitigation decisions based on the observations from production.

Building Narya to work for a production cloud requires both algorithm designs and systems support. We first describe the core prediction and mitigation algorithms in Section 4 and Section 6, respectively. Section 7 describes the Narya systems design and implementation.

3.2 Key Optimization Metric

Narya's objective is to reduce and minimize the overall customer impact caused by node failures on the fleet. Defining a good cost metric for customer impact is critical for the Narya's decision engine to optimize that metric. In Azure, we focus on the Annual Interruption Rate (**AIR**) defined as:

$$AIR = \frac{\text{VM interruption count in } T}{\text{Total VM lifetime in } T} \times 365 \text{ days} \times 100 \text{ VMs}$$

T is any given measured interval duration in days. VM interruption in this paper mainly refers to reboots or loss of heartbeats. Internally, we also measure performance drop with a sub-metric we call *AIR-blips*.

We optimize this metric instead of the traditional availability metric for a couple of reasons. First, long-duration incidents are now rare in Azure. VM interruptions become more common that require addressing. Second, short VM interruptions can significantly disrupt user experiences, e.g., for gaming-type applications. Third, for VMs that run applications like databases, even if the VM only experiences a short interruption, the applications take time to recover, which translates into a longer user-perceived interruption. Fourth, based on communications with customers, customers can be more annoyed if their VMs get frequently interrupted when compared to a single longer-time interruption.

3.3 Challenges

We need to address several design challenges. First, failure mitigation has to act with incomplete information since the underlying root cause is not yet known. For Narya, this challenge is even more pressing since the failure has not occurred.

Second, due to the massive scale of a cloud system, there are many factors to consider in the decision logic. A decision may work well for some nodes but not others. If not careful, some corner cases can mislead or bias the decision logic. Narya must be robust enough while still being flexible.

Third, our experience suggests that when incorporating failure prediction into a production cloud system, false prediction is unavoidable due to the complex system environment, large number of noisy signals, unexpected customer workloads, etc. If the system blindly trusts the failure prediction results and reacts, it could cause unnecessary disruptions. When consuming the prediction results, the mitigation mechanisms should take this into account and operate in a way that minimizes the impact due to the unavoidable false predictions.

Lastly, failure mitigation is a mission critical procedure. If not designed well, a decision engine may do more harm than good. Ensuring safety should be a top priority for Narya.

4 Predicting Node Failures

The first step in Narya is to predict a host failure before it occurs. In this Section, we describe two prediction methods Narya uses: (1) static threshold rules written by domain experts; (2) machine learning model-based prediction.

4.1 Input Signals

Narya consumes telemetry signals from the entire stack of the host environment to make informed prediction. For hardware and firmware, the monitoring agents collect low-level logs from disk SMART attributes, memory (*e.g.*, uncorretable errors), CPU (*e.g.*, machine check error), motherboard (*e.g.*, bus error), *etc.* A higher-level source of signals comes from device drivers, *e.g.*, timeout events. Repetition of such events is often an indicator of an imminent failure. Faults in individual component do not necessarily cause customer impact. Some could be transient that would go away after retries. Others may be tolerated by redundancy. Narya further consumes critical OS events and aggregate application performance counters.

Another important source of signals used by the predictor are results from the control-plane operations. For example, repetitive VM creation operation errors could indicate serious host issues even if the host still appears to be running. Such signals help reduce the observability gap [16].

4.2 Rule-based Prediction

Rule-based prediction leverages domain knowledge from hardware, firmware and software experts. We analyze the common failure patterns and the available telemetry signals to predict failures that have significant customer impact. For example, in some cases, CPU Internal Error (IERR) is a good indicator that the node will fail *again* soon; a prediction rule could be marking the node if IERR occurs twice within 30 days. Rules are typically written as Json files or Python scripts

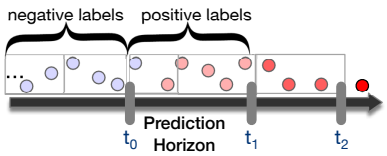


Figure 2: Prediction horizon and label timeline. t_1: host failure time. t_2: component permanent failure time.

or sometimes C++. Since rules are manually written, they are simple and easy to understand. The prediction rules are deployed directly in the host and can be executed fast.

Rule-based prediction works best for definitive signals that indicate some severe issue with high confidence. An example is the `AvailableSpare` signal in NVMe device health log. When it drops below a certain threshold, we know the device is almost at the end of its life.

Since many failure signals are not definitive, rule-based prediction cannot cover a wide range of imminent failures. Without leveraging different signals, it generally does not provide enough lead time to mitigate before the failure occurs. The number of prediction rules also keeps growing, which becomes a burden to manage and tune. We have a total of 51 rules in use in Azure compute.

4.3 Learning-based Prediction

To address the limitation of rule-based prediction, Narya employs an additional learning-based predictor, which analyzes more signals and patterns during larger time window. It can predict many complex host failures. It also can predict further ahead of time, thus leaving longer time for the mitigation engine to take actions. In comparison, static prediction rules are usually only triggered close to actual failures.

Prior work predict disk failures [14, 27, 38] and node faults [25] with supervised learning. Our learning-based prediction aligns with prior solutions. A main difference is that we focus on overall host health and failures that result in customer impact, instead of failures of individual components. Because of this, Narya analyzes more diverse signals across layers such as control-plane operation signals.

Prediction Horizon and Label. Deciding the labels to use for learning is crucial for Narya. In prior work that predicts hardware failures for alerting, the positive labels are signals close to when the hardware is completely broken. For Narya, the host view of a failure is different from individual components' view. The host failures could be could be unresponsive host, VM creation failure, host OS crash, *etc*. In our observation, they happen much earlier than the permanent failure of a component (e.g., disk unusable). As Figure 2 shows, if we assign positive labels from time t_2 (permanent component failure), it can yield late prediction which comes after host failure at time t_1. The consequence is that the prediction does not give enough time for Narya to take proper mitigation actions.

Additionally, certain faults might not be a problem to the source component but could be problematic from host's view.

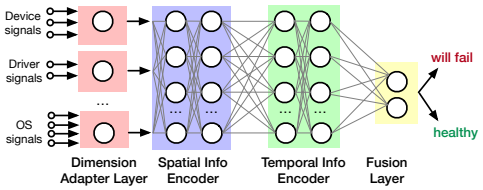


Figure 3: Deep learning model structure.

For example, a series of memory correctable errors might seem fine for an ECC DRAM because they are corrected. But the host may already suffer slowness and impact VMs.

To get accurate and useful prediction result, we only use host failures that result in customer impact and are later confirmed to be caused by some hardware component faults during diagnosis. For a given host failure, if it occurs at time t, we assign positive (failure) labels for signals from $t-1$ to $t-n$, where n is the prediction horizon and using an hour unit. We assign negative (normal) labels for signals from $t-(n+1)$, ... We also sample negative labels from healthy nodes.

In production, our prediction horizon is set to 7 days. We made the choice based on how discriminative the feature will be given different horizons. Specifically, we looked at the feature distribution of failed nodes and measured the same distribution of healthy nodes. We then measure the similarity between the two distribution groups. Beyond 7 days, we could not observe a significant difference.

Machine Learning Model. With the signals, labels, and host metadata, Narya trains a binary classifier. The predictor outputs the failure probability of a host (we use 0.5 as the cutoff).

To train the classifier, we use the gradient boosted tree model [18] commonly used in supervised learning, which combines decisions from a sequence of simple decision trees with a model ensembling technique called gradient boosting [10]. This simple model works fine for our scenario in terms of its predictive power. But we have to carefully craft approximate features from the signals for it. We engineer 2k+ features from 100+ time series data. Those engineered features are combined with other categorical features to build the learning-based model feature set.

We further explore reducing the feature engineering efforts by directly learning the features with an attention-based deep learning model [20, 35]. At a high level, we aim to learn both spatial features and temporal features. Spatial features compare one component to its neighbors. For example, one host often has multiple disks configured under RAID 0, thus they are expected to perform similarly. If one disk performs worse than its neighbors, it could indicate imminent host failures. Attention-based deep models are designed to capture such patterns so that more weights (attention) are assigned to anomalous neighbors. The temporal features characterize changes in components over time.

Figure 3 shows the structure of this model. First, we use a dimension adapter layer to unify the dimension of signals from different sources. Next, we employ a spatial information

	Action	Description
Primitive	Avoid	Deprioritize new VM alloc. on this node
	Unallocatable (UA)	Block new VM allocations on this node
	Live Migration (LM)	Migrate VMs to other nodes on the fly
	Service Healing (SH)	Discon. VMs, move them to healthy nodes
	Soft Reboot (SR)	Reload host OS kernel, VM states preserved
	Human Investigate (HI)	Shut down node and send it to diagnostics
Composite	UA-LM-HI	Block alloc., attempt LM and HI after T
	UA-SR	Block alloc., attempt soft reboot
	UA-LM-RH	Block alloc., LM and unblock after T
	Avoid-RH	Avoid alloc. to this host if possible

Table 2: Primitive and composite mitigation actions for Narya. Composite actions are sorted by decreasing priority.

encoder based on self-attention. It calculates weights of a component's neighbors. The weighted sum of the neighbors' feature vector represent its spatial information. Then, we use the temporal information encoder, which consists of positional encoding, self-attention, and location-based attention layers. Finally, we employ a fusion layer to do binary classification. We omit the attention implementation details as they based on an existing technique proposed by Lee *et al.* [20].

Overall, this model achieve 5–10% improvement compared to the decision tree model with hand-crafted features.

5 Mitigation Actions

When a host is predicted to fail, Narya chooses among several possible actions. Table 2 lists the main *primitive actions* in Azure. Mitigating a failure often requires multiple primitive actions. An aggressive goal for Narya is to explore the actions arbitrarily and figure out the optimal combination. But this could potentially bring significant customer impact. Instead, Narya mitigation engine focuses on exploring pre-defined *composite actions* (Table 2). This set of composite actions is constantly enriched with new combination and by modifying parameters (e.g., unallocatable duration).

Live Migration moves a running VM from one host to another with minimum disruptions. The migration process involves transfer of the VM's memory, processor and virtual device state [7]. The LM engine iteratively copies the VM's memory pages while maintaining a dirty page set for the VM on the source host. Based on the dirty page rate, network bandwidth, the engine determines the maximum iterations to stop the VM. After the VM is stopped, the LM engine synchronizes the dirty state with the target and resumes the VM on the target host. Note that not all VMs are eligible for LM and LM could fail for various reasons.

VM Preserving Soft Reboot preserves the VM state across a reboot of the host OS. At a high level, the host OS kernel is reloaded into memory, the VM memory and device state are persisted to the newly loaded kernel. The host reboots into the loaded kernel while preserving the persisted state. Once the reloaded kernel starts, the persisted state is restored and the rest of the state in the prior kernel are discarded. The restored VM experience a brief pause similar to live migration.

Service Healing is used to restore the service availability of unhealthy or faulted VMs. Live Migration can move running VMs transparently, but it could fail or cannot be applied due to certain constraints such as network boundary. Service healing works for more general scenarios. The VMs will be isolated by powering down or disconnection from network. The controller generates new assignment of the VM to healthy nodes. During the process, there is some interruption.

Mark Unallocatable blocks allocation of new VMs to a host for some time T (default 7 days). Composite actions typically start with marking a suspected host unallocatable. In *UA-LM-HI*, after marking host unallocatable, the controller attempts to live migrate the VMs on this host to other hosts. After all VMs have been migrated or destroyed by customers or this host fails, the host will be sent to diagnostics. If at the end of the unallocatable period T some VMs are still running (*e.g.*, because they are not eligible for LM) we service heal them before pushing the host to diagnostics. *UA-LM-RH* is a variant of *UA-LM-HI* where we unblock allocation (reset node health) at the end of T. In *UA-SR*, the controller blocks the allocation and then try the kernel soft reboot action. If the soft reboot succeeds, the controller unblocks allocation. Otherwise, we use a fallback strategy, typically *LM-HI*.

Avoid informs the allocator to try hard to avoid adding new VMs on this host. The motivation is that blocking allocation has a strong impact on capacity since the host is not eligible for getting new VMs. Thus, the number of hosts that can be marked unallocatable at the same time is limited. *Avoid* action provides a weaker constraint. In addition, it will not perform live migration. The behavior on host failure is still to send it to diagnostics. At the end of T, we reset the node availability.

NoOp is a special action for predicted failure, in which the controller does not take any action. This is the baseline to measure the benefits of prediction and taking actions.

6 Decision Logic for Adaptive Mitigation

With different prediction rules/models as well as different mitigation actions, relying on static assignment based on domain knowledge to map each prediction to an action can soon get intractable and ineffective. This motivates the design of Narya decision engine for adaptive mitigation.

6.1 Online Experimentation with A/B testing

One straightforward way for choosing mitigation action is to *estimate* offline the impact for each possible action for a predicted failure. In our experience, given the complexity of cloud systems, it is extremely hard to estimate the impact of actions and know which one is good without trying them in production. Based on this insight, Narya takes an online experimentation approach to evaluate the effectiveness of different mitigation actions by testing them at scale.

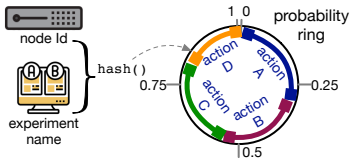


Figure 4: Hash node ID and experiment name for sticky assignment.

A/B testing, also called online experiments, is widely used [19] in front-end designs to test the effect of UI features. Narya adopts the A/B testing methodology and adapts it for discovering good mitigation actions. In classic A/B testing, one experiment is about one UI feature and each unit is a user. For Narya, one experiment is about the mitigation of one failure prediction (*e.g.*, CPU IERR, slow memory access latency), and each unit is a failure mitigation request about a node marked by the corresponding prediction rule/model.

The workflow of the A/B testing in Narya is as follows: (1) each predicted node is assigned to different action groups with equal probability (2) after taking each action, we measure the customer impact within an observation window; (3) we use hypothesis testing to test if an action yields significantly less customer impact than others; (4) once statistical significance is reached, we consider this least-impacting action optimal and apply it for all nodes; (5) we keep monitoring the customer impact per node for the used action; (6) if customer impact significantly increases, we run a new A/B testing experiment to validate that the action is still optimal.

Cost. The cost ($= -1 \times reward$) models the customer impact. This cost needs to include as much as possible the different trade-offs in our system. It is critical that the pros and cons of each action are modeled into the cost to be able to correctly optimize for the action to take. We use the number of VM interruptions in the node during an observation window. In addition, we add the VM interruptions in nodes to which we migrated VMs as part of live migration or service healing.

An additional constraint we need to consider is capacity. As capacity does not directly impact customer and is not visible at the node level, it cannot be easily added into the cost. Our current strategy to incorporate that constraint is to limit the total number of nodes that can be marked unallocatable for the same rule in the same cluster at the same time. Doing this, we can indirectly impact cost of marking nodes unallocatable.

Assignment Strategy. A crucial point for A/B testing is to decide for each node marked by the failure predictor, in which experiment group should it go to. In classic A/B testing, each experiment unit is assigned randomly, based on the assumption that the units are independent and identically distributed (i.i.d). For Narya, we make several changes.

The same node can be marked by the same prediction rule multiple times during an A/B experiment. In this case, if Narya assigns it different mitigation actions, e.g., assigns node X in action A group at time t_1 and then to action B at time t_2, the i.i.d assumption can be violated. This is because the underlying node condition at t_1 and t_2 could be highly correlated, especially for hardware issues. Then the observations from the treatment and control group are correlated.

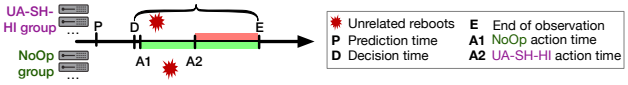


Figure 5: Using decision time as the start of observation window.

To address this issue, we introduce **sticky assignment**: if a node is assigned to take action A in an experiment, for subsequent mitigation requests for this node during this experiment, the engine will assign the same action by using hash of the node Id and experiment name (Figure 4).

Classic A/B experiments are typically done sequentially. In our case, sequential experimentation takes too long; so Narya allows different experiments to take place concurrently. While most experiments are independent, some experiments could have prediction rules that are correlated. In this case, it is important to test all possible action combinations to analyze their compound effect later. For example, with experiment X testing actions `{a,b}` and a correlated experiment Y testing actions `{c,d}`, we need to have observations that take each of the four scenarios: `(a,c)`, `(a,d)`,`(b,c)`, `(b,d)`.

Action Overriding. Since many fault handling policies, including our A/B experiments, can take place concurrently, a host can potentially be marked by several prediction rules. As a result, the host might need to follow different composite actions at the same time. A common reason for this is the incomplete information factor (Section 3.3). To handle this situation, Narya uses a specific override logic based on the priority from the order in Table 2. When we try to assign a node with a lower-priority action than its current one, we skip it. In case of equal priority, Narya honors the older actions. The rationale is that later prediction can often be a side effect of the earlier one. Since we often do AB testing between actions with different priorities, it is critical to observe cases where the action was skipped or later overridden.

Effect Observation and Attribution. Depending on the complexity of the actions, some need longer time to get triggered. In the time between the decision is made and the action is started, VM interruptions can happen and are not caused by the action. However, to fairly compare action, we should still count the cost in this time gap because, for instantaneous actions, it would be impossible to differentiate the costs caused by the action from the costs that would have happened anyways. For this reason we use the decision time instead of the action time as the start of the observation window. Figure 5 shows an example where if we used the actual action time we would ignore some uncorrelated reboots for one action and not the other, while they happen in both groups.

Hypothesis Testing. After collecting the effect metrics about experimented mitigation actions, the decision engine performs hypothesis testing to decide whether an optimal action is discovered and if the experiment can be stopped. Since our cost function is complex and depends on external variables, we

simplify the hypothesis testing by assuming that the number of VM reboots per node is i.i.d and follows a normal distribution. Since different actions can highly change the VM reboots per node, we will not assume equal variance for the different actions. Under these assumptions, we use Welch's t-test [36] when testing for two actions and Welch ANOVA test for 3 and more. In the latter, we use post-hoc analysis to remove all statistically worse actions until one action remains.

6.2 Bandit Modeling

One drawback of A/B testing is its static group assignment. Before statistical significance, we do not leverage the estimated difference between the groups to minimize our cost, and once an experiment reaches statistical significance, we will almost always use the discovered optimal action. This is essentially the classic *exploration-exploitation* dilemma. Naturally, we explores modeling the adaptive mitigation as a *Multi-Armed Bandits* problem [32, 33], where we aim to minimize the customer impact over time by ensuring a balance between continuously exploring potential better actions and exploiting the discovered best action. At training time, we observe tuples `(rule, node, chosen action, chosen action probability, cost)` to estimate the probability to choose each action, while at serving time, we match a request tuple `(rule,node,timestamp)`, to the learnt action.

Actions. The output of the bandit model will be which composite action we want to attempt. The available actions are typically defined per experiment based on offline analysis of the prediction signals. The most important estimation needed are the false positive ratio, the time to failure/impact and the feasibility of our main mitigation actions (Section 5).

Exploration Algorithm. To minimize customer impact over time, we face the classical exploration-exploitation trade-off. We need to explore different actions to see which one minimizes the customer impact but at the same time we want to use the action with minimum estimated cost as much as possible. In other words, we need to balance between short-term and long-term benefits. We experimented with multiple different exploration models including Epsilon Greedy and UCB, and decided to use *Thompson Sampling* model since it provides more explainability and continuous probability changes. In Thompson Sampling, we model the reward as a function of actions and some model parameter and choose the action according to the probability that it maximizes expected rewards. This Bayesian approach updates the prior using observations of actions taken and chooses each action with probability equal to the chance that it minimizes the expected cost:

$$P(a*) = \int I\big(E(c|a*,\theta) = min_a E(c|a*,\theta)\big) P(\theta|obs) d\theta$$

where $P(a*)$ is the probability to choose action $a*$, θ is a hidden parameter, c is the cost and *obs* are the past observation as list of tuples (a_i, c_i). Appendix **??** describes in more detail the Thompson Sampling algorithm in Narya.

6.3 Extension to Bandits

Compared to traditional Bandits, our system faces several challenges. In addition to the effect observation solution described in Section 6.1, we made 4 main adaptations as follows.

Accommodate Temporal Changes. Since our system can change in time, older observations will gradually become less and less relevant. To account for this factor, we use an exponentially decaying weight for observations to focus on recent data. We will apply a multiplying weight to past observation in the format of $decay = \sigma^{T-T_{obs}}$, where σ is the decaying factor, T is the current time and T_{obs} the time of the observation. We set σ by default to 0.99 based on simulation-based experiments and so that the weight would be close to 0 after 3 months which is our typical retention policy. In the case of Thompson Sampling with Gamma Prior, the distribution to sample from becomes:

$$P(\theta|a,obs) \sim \Gamma\left(1 + \sum_{i,a_i=a} c_i \sigma^{T-T_i}, 1 + \sum_{i,a_i=a} \sigma^{T-T_i}\right)$$

Delayed Reward Collection. A key challenge in our settings is the potential long time between the action taken and its impact. This forces us to observe for at least 10 days and up to 30 days the impact of choosing each action. This observation window highly depends on the duration of the action and its effect: UA-LM-HI for 7 days would require around 10 days while Avoid-RH for 15 days would require a full 30 days to observe potential failures following the reset health action. The drawback of a long observation window is the delay for the reward to be integrated into the model. Thus, wrong estimation could be used for a while before the observed cost can readjust these probabilities. One way to counteract this effect is to observe the reward as it comes. But we can suffer from the opposite effect of getting biased by reboots close to the decision time. Our experience suggests that we need to wait for the full initial observation window and then can collect partial rewards incrementally.

Bandit stickiness. Since the probability to choose each action over time changes, we cannot rely on a hashing function used in our A/B testing to make sure a node will be assigned to the same action in case of repeating prediction. We define the bandit stickiness for time T as reusing the previously chosen composite action if the node has an available decision for the same rule within the T time window.

Deal with Unexpected Spikes. Another potential issue in our system is the unexpected spike of VM interruption events that could affect one action group more than the other. One approach would be to perform an outlier removal step before using such observation, but in that case it could also filter out spikes inherent to a specific action, which should be integrated into our learned model. We address the issue with the safe guards mechanism described below.

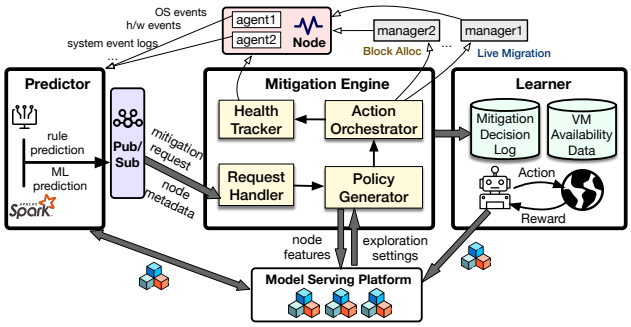


Figure 6: Narya system architecture, which consists of the predictor, mitigation engine and learner. The ML models for the prediction and mitigation are stored in a model serving platform [8].

6.4 Safe Guards

Safety is a top priority in Narya mitigation decision logic. We take several measures to ensure safety. The action overriding priority (Section 6.1) is one measure. We also apply *safety constraints*—domain-specific restrictions to prohibit certain actions in some failure scenarios. In addition, Narya decision engine requires a minimal number of observations before following the recommendation from the Bandit. The Bandit model will output a *premature* flag for insufficient observations, in which case we would fall back to the default action probabilities that are given similarly to A/B testing. This also helps dilute the potential effect of spikes in a larger observation set. Moreover, we support configuration of minimum and maximum constraints for each action probability. The maximum constraint limits the possible reactions to high cost, while the minimum constraint guarantees some exploration for actions that could seem irrelevant at a specific time. In practice, any experiment for which we expect a potential change in the system should keep a minimum probability for each of the allowed action so that it will continue observing the effect of such action. Experimentally, We found that using 10% exploration when nodes flagged per day is less than 100 and 5% when it is higher yielded the best results.

7 Narya System Design and Implementation

In this Section, we describe the system support for Narya. Figure 6 shows the system architecture. Narya is deployed in each data center region of Azure compute. The Narya system must be able to process the massive signals and requests from the entire fleet with low latency and reliability.

7.1 Failure Predictor

Azure deploys various agents in each node to monitor the health of the host environment. The Narya predictor ingests health signals from these monitoring agents and runs rule-based prediction and ML-based prediction (Section 4).

Rule-based prediction has low cost and high priority. Thus its prediction logic is executed directly in a host. The ML-based prediction inspects much more signals such as performance counters and runs more complex prediction logic. Thus, the ML predictor is implemented as a centralized service. It collects raw signals from monitoring agents using micro-batches (small groups) and incrementally processes them. Open source technologies are used for ML modelling. LightGBM is used for decision tree model and PyTorch for deep learning model. The ML inference tasks run as an hourly Spark job, which reads the most recent signals and the trained ML model to compute failure probability for each host.

```
"HW_Triage": {
  "Type": "Selection",
  "ChildSelector": [{
    "ShouldSelect": "C#|Request.FaultedHwHealthGrade == 100",
    "Child": "HW_Try_HI"
  }, {
    "ShouldSelect": "C#|Request.FaultedHwHealthGrade == 75",
    "Child": "HW_Try_Unallocatable"
  }]
}
"HW_Unallocatable_WithRecovery": {
  "Type": "Action",
  "Actions": {
    "MarkNodeUnallocatableAction_WithRecovery": {
      "Action": "MarkNodeUnallocatableAction",
      "Input": {...}
    }}
}
```

Figure 7: Example of mitigation policy tree nodes.

Pub/Sub Service. A *mitigation request* is created if a node is predicted to fail with high probability. The predictor publishes the request along with metadata information about the host (e.g., hardware generations, OS version) to a central pub-/sub service, which we implement on top of Kafka [1]. We choose Kafka because it allows scalable, low-latency, and real-time streaming processing to deliver the mitigation requests quickly. Also, our computation pattern is many data producers with a small number of consumers, which is a main target scenario Kafka is designed for.

7.2 Mitigation Engine

The mitigation engine is a core component of Narya. Internally, it is composed of four major microservices. These microservices communicate with each other and other services in Azure using REST APIs.

Create Mitigation Job. The *Request Handler* microservice consumes mitigation requests from the Pub/Sub service. Upon receiving a mitigation request, the request handler creates a mitigation job with a job Id. This job Id is used by other micro-services to track the mitigation and query its progress.

Instantiate Mitigation Policy. For a new mitigation job, the *Policy Generator* needs to create a *mitigation policy*. A mitigation policy maps from information in a mitigation request to the action to take. It is represented as a decision tree. There are two types of tree nodes: a `Selection` node, which chooses the tree node to visit next based on some C# predicate; an `Action` node, which executes a user-defined C# function. The decision tree structure allows us to easily specify the decision logic. For example, we can decide mitigation actions based on failure types (software or hardware), fault codes (e.g.,

`Request.FaultCode == 0x123`), cluster types (storage or compute), hardware generations (e.g., `HostNode.Gen != "ABC"`), etc. Figure 7 shows an example.

The policy is derived from a template (Json configuration file). For A/B testing, the action distribution is specified in the template. In the `Action` tree node matching a mitigation request, the node's C# function samples one mitigation action from the distribution. For Bandit, the `Action` tree node dynamically generates the distribution based on contextual information. In particular, it calls a ML model serving platform, Resource Central [8], with relevant features such as the fault code, VM count, etc. The platform returns an exploration setting—a probability distribution over mitigation actions.

The policy generator then applies safety constraints on the retrieved exploration setting to obtain an adjusted action probability distribution. Additionally, the mitigation policy allows imposing *rate limit* for a tree node to avoid excessive mitigation that could cause capacity issue or cascading failures.

Walk Policy Tree. The policy generator further traverses the policy tree in DFS order and creates an *action plan*. During this process, the generator performs many steps such as checking predicates, checking rate limit condition etc. If a node is entered, the generator first checks if we need to apply *sticky* mitigation action (Section 6.1). If so and there were decisions made within certain period of time for the same experiment and tree node, the last mitigation action is retrieved. The decision history is persisted to a distributed storage service.

If no previous decision is found and the entered tree node is run in an A/B testing mode, one action is sampled based on the probability distribution. If the entered tree node is in Bandit mode but there is insufficient data learned, a specific flag is returned to indicate the Bandit model is pre-mature. Then the generator falls back to use the action probabilities from A/B testing mode. This allows us to bootstrap bandit learning from A/B testing safely, especially considering the delayed cost feedback loop. In addition, if there is any error in calling the model serving platform, the generator falls back into A/B testing mode to ensure safety.

Carry out Action Plan. The *Action Orchestrator* microservice is responsible for carrying out the action plan from the policy tree walk session. This step involves making API calls to the corresponding compute managers since different actions may be implemented by different managers. The orchestrator executes actions asynchronously to avoid blocking.

Log Actions. Logging in general is very important for data analysis, Bandit training, and counterfactual evaluation of different mitigation policies. The logging format for Bandit learning is special since it requires not only recording the chosen mitigation action but also the probability. In particular, the mitigation engine will log the action timestamp, experiment name, model type, model name, model version, action distributions, chosen action, chosen action parameters, etc.

Track Node Health. The *Health Tracker* tracks node and

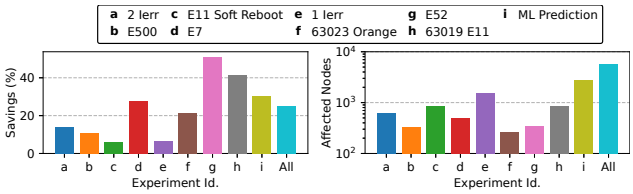


Figure 8: AIR improvement of all A/B testing and Bandit experiments in March 2020, breakdown per experiment.

VM health information during the mitigation process. For example, while rebooting a node, if we get a new signal (e.g., a WindowsEvent) that it is a hardware issue, then we can HI the node early instead of waiting for reboot to fail/timeout.

7.3 Learner

Learner is a centralized component in Narya. It learns the effect of mitigation action across different datacenter regions. Compared to a regional learner design, global learner has the advantage of observing more data points and hence more confidence in the cost estimation. Additionally, the mitigation effect change in certain region due to software/firmware updates could be quickly learned and applied to other regions rolling out the same updates.

The learner runs two main jobs: cost collection and Bandit model training. The cost collection job retrieves the mitigation engine's decisions from the logs. This information is then correlated with the VM availability measurements and other important information (LM status, VM workload, etc.) to determine the cost of the mitigation action for training. The Bandit model training runs on a Spark cluster. The output model of the learner is a categorical distribution, which the model server can easily draw samples from.

8 Evaluation

Narya has been running in production since June 2019 to prevent VM interruptions in Azure compute platform. Our evaluation answers several questions: (1) how effective is Narya in averting VM interruptions? (2) how accurate and timely is the failure prediction? (3) how does Bandit model compare with A/B testing?

8.1 VM Interruption Savings

The main metric we use to evaluate the effectiveness of Narya is the VM Annual Interruption Rate (AIR) (Section 3.2). We measure the delta between the AIR using the old static assignment and the AIR under new mitigation decisions from Narya. We specifically compute three metrics: the estimated daily AIR savings, the oracle daily AIR savings (savings if we already knew what was the best action), the regret (how much additional AIR we could have saved). The estimated daily AIR savings ($\hat{S}$) is obtained by comparing the impact of each tested action to the impact of the original action projected on the whole fleet. The oracle daily AIR savings (S^*) is

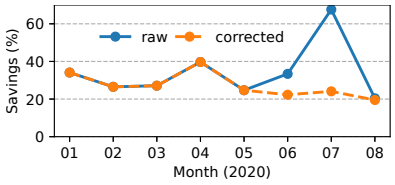


Figure 9: AIR improvement per month.

estimated by mapping the best performing action to the whole population compared to the original action on the whole fleet. Appendix **??** shows the formulas to calculate $\hat{S}$ and S^*. The regret is the expected difference between the reward sum associated with an optimal strategy and the sum of the collected rewards. We consider $R = S^* - \hat{S}$ to be our AIR regret, meaning how much additional AIR we could have saved if we knew the best action all along.

Due to the confidentiality nature of AIR, we report $\hat{S}, S^*, R$ as relative percentages compared to the overall AIR contributed by all of our target failure types (host failures caused by various hardware problems). For the month of March 2020, $\hat{S}$ **is a 26.2% improvement**, i.e., Narya successfully decreases AIR by about 26.2% compared to the static strategy. We also provide the per-experiment improvements in Figure 8. This shows, for each AB or bandit experiment what percentage of VM interruptions were prevented compared to using the original strategy. 63019E11 constitutes the largest savings since the AB testing experiment was already using the best action. For the same period, S^* is 35.4%, meaning the best AIR reduction percentage we could possibly achieve (if we use the optimal action); R **is 9.2%**.

8.2 Savings Trend Over Time

Figure 9 shows the AIR improvements over time. Overall, the savings fluctuate between 20% to 40%. July saw a sudden jump. This is because one major firmware fix occurred in the June-July time period. One rule started marking much more nodes, including nodes considered false positives (not likely to fail soon). This largely increased $\hat{S}$ as the old policy was very sensitive to false positives. However, our anomaly detection has caught this issue. We probably would have fixed the policy if we were using it. We report in Figure 9 the corrected savings assuming that fix. In addition, this firmware deployment fixed a driver issue for which our mitigation was reducing much AIR by predicting failures in advance. As a result, our savings decreased in July and August.

8.3 Accuracy and Timeliness of Prediction

We first measure the precision and recall of the Narya failure predictor. There are multiple rules or models to predict different types of host failures. For a prediction rule/model r, we define the prediction precision as $\frac{F+D}{N}$, where N is the total number of hosts marked by r; F is number of hosts that fail and are diagnosed to be indeed caused by the suspected fault; D is number of hosts that Narya successfully mitigate and the suspected fault is later confirmed. The recall is defined as $\frac{F+D}{M}$, where M denotes the number of host failures that are diagnosed to be caused by fault type that r represents.

The overall precision is 79.49%, while the overall recall is 50.7%. While the false positive rate (20.51%) is not small, we note that failure prediction in a large-scale, complex, and frequently changing cloud like Azure is an extremely challenging problem. Narya is designed with the expectation that false prediction is unavoidable and employs several measures such as action overriding, safety constraints, longer observation window to minimize the impact of false prediction.

We further evaluate the contribution of different signals (features) to the prediction accuracy. We calculate the feature importance using the SHAP method [28], sort features by their importance, and group them into 10 bins. We then evaluate the precision and recall (F1-score) using individual bins or aggregate bins (features from bin #1 to #N). Figure 10 shows the result. We can see that some features are more important than others. The first bin in particular contributes significantly. Examples of features in the first bin include read error rate, flush count, AvailableSpare, HostReadCommands, etc.

Besides precision and recall, for Narya, how timely the prediction result is also crucial. Prediction time to failure (or lead time) measures the duration between a server is predicted to fail to the time it fails. A larger lead time gives Narya enough time to take preventive actions. Figure 11 shows the CDF of the ML prediction time to failure. ML prediction provides a median lead time of 2.44 days. Figure 12 compares the timeliness between the ML-based prediction and rule-based prediction. We can see that ML-based prediction provides significant advantage in the timeliness.

We further measure the quantitative benefit of early prediction to live migration success. For nodes predicted to fail, the average successful live migration per node is 5.57. In comparison, for nodes predicted to fail but with small lead time, the successful live migration per node is 3.

8.4 Comparing AB Testing and Bandit

Next we compare Narya Bandit and A/B testing in finding the optimal mitigation action. To do so, we compare the used strategy to the other possibilities. For mitigation requests that go through A/B testing, we use off-policy learning of Bandit based on observed A/B data. Similarly we compare the Bandit output to a static A/B policy. We then use counterfactual estimation [32] with Inverse Propensity Score to estimate the cost of running Bandit in place of A/B testing. Over the 2 months period of February and March 2020, using Bandit instead of A/B testing for ongoing experiments could have helped decrease the number of VM interruptions by 14.4%. The breakdown per experiment can be found in Figure 13. Note that A/B testing compared to bandit provides more safety and the possibility to look at more than on cost metric.

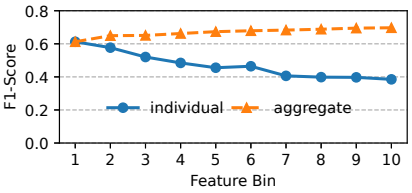


Figure 10: Contrib. of different features.

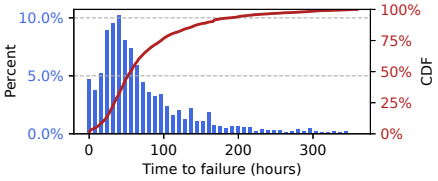


Figure 11: ML prediction time to failure.

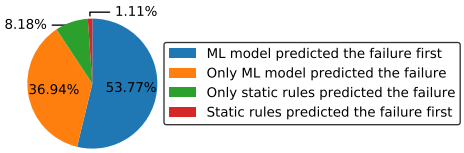


Figure 12: Prediction timeliness.

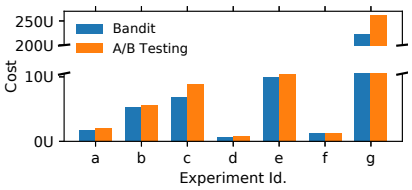


Figure 13: Cost metric (VM interruptions, units anonymized) under Bandit vs. AB testing.

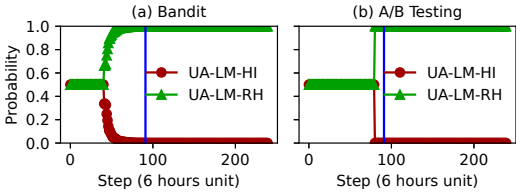


Figure 14: Median probability of choosing each action using Bandit and A/B testing over 1000 re-sampling simulations based on production data.

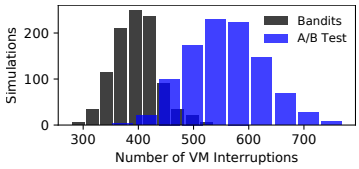


Figure 15: Distribution of reboots using Bandits or A/B testing in 1000 simulations based on production data.

Min	Median	Max	No Convergence	Ongoing
12 days	29.5 days	140 days	2 experiments	3 experiments

Table 3: AB testing experiments convergence time.

8.5 Convergence to Optimal Action

Table 3 shows the convergence time of A/B testing for different experiments. The variance of the convergence time is big, because we need to accumulate enough samples. For different rules, the time it takes to collect the samples differs a lot. Two experiments did not reach convergence at the end of experiments. No convergence usually indicates that there is no significant difference among the experimented actions. For Bandit, we compare its behavior with A/B testing through simulation with production data (Appendix **??**). Bandit can achieve a much faster convergence to the best action. Figure 14 shows the result. Bandit converged in around 50 steps, while AB testing would converge in 125 steps. Faster convergence also yields more AIR savings. As Figure 15 shows, Bandits yields much fewer reboots than AB testing.

8.6 Case Studies

Blobcache error is a symptom that can caused by hardware issues or some recoverable faults. The original mitigation policy was *UA-LM-HI*. We wanted to test if we could avoid the impact of service healing at the end of the unallocatable period and try to reset node health (unblock allocation) instead. We used an AB testing experiment to compare the VM reboots per node when using *UA-LM-HI* and *UA-LM-RH*. The experiment started on 03/10 and statistical significance was reached on 03/25 and observed on 04/02. We were able to save 25.2% AIR compared to the old policy during this period. Once, we adopted the new policy (config change to make it 100%) and deployed it to production, it saved 50.3% of AIR associated with this type of failure.

E11 is a Windows event indicating a disk controller error. When this event occurs, it generally means that the hard disk is

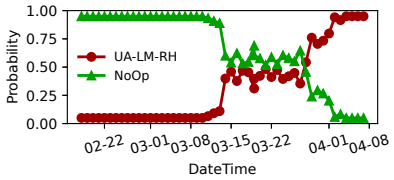


Figure 16: Action probability change using Bandit decisions.

experiencing some issues most likely indicating an imminent failure. Using offline correlation analysis on non-empty nodes with no prediction from other rules/models, we found that 85% failed in the following 7 days, with a majority in the first few hours. Although the lead time was small, it should have sufficed to migrate some VMs in each of these nodes.

We started our first AB testing experiment on 2019-07-25 to try our *UA-LM-HI* policy over *NoOp*. To our surprise, we did not see significant impact as we imagined. Upon analysis, it appeared that few of the VMs could live migrate, mostly because the node failed to quickly or the disk state was already too bad to succeed in live migrating the VMs. However, after a few month of AB testing, the difference in reboots revealed to be significant mostly through short lived VMs getting stopped and repeated failure being avoided by the unallocatable policy. The experiment was ended on 2019-12-12 with approximate daily AIR savings of 28% for such signature.

Another AB testing experiment was started 2020-01-11 to test the potential saving of using soft reboot to mitigate some of these issues, following a few positive offline test of that action. Contrary to our belief, we observed no significant improvement, even a slightly larger VM reboot per node in the *UA-SR* action. In total, 51 nodes tried the *SR* primitive action, with none of them succeeding because some pre-checks were not met. This shows the importance of properly testing all potential new actions before using them on the whole fleet.

I/O Timeout We started an AB experiment between *UA-LM-RH* and *NoOp* actions on 2020-01-11 for an I/O timeout prediction rule. At first, *NoOp* seemed to be the better option, although not significantly. In late March, the *UA-LM-RH* action started being consistently better and led to us to switch to

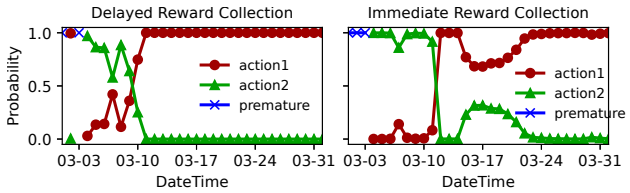


Figure 17: Probabilities in taking two actions using delayed versus immediate reward collection. Action1 is optimal.

Bandit. As Figure 16 shows, we can see a clear switch from choosing *NoOp* to choosing *UA-LM-RH* for that time. Even though we are unsure as to what system changes trigger the probability change, our Bandit model picks up the changes and adapt. Using counterfactual estimation, the results show the Bandits adaptation yields savings of 3.7%–13.9% compared to both static policies.

8.7 Reward Collection Schemes

We compare three possible reward collection schemes: (a) delayed reward collection; (b) immediate reward collection; (c) incremental reward collection.

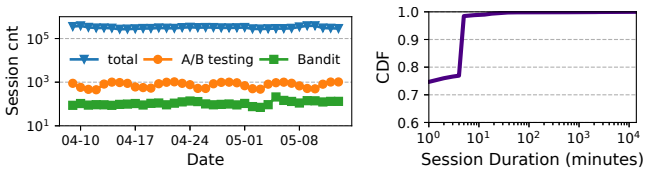
Delayed reward collection (§6.3) is our default scheme. In (b), we associate VM interruption costs with an action on the day where the action is completed, and we update the Bandit model. In (c), we update and associate the cost daily.

Across different A/B testing experiments we run, the three schemes have varied effectiveness. But overall delayed reward collection outperforms immediate scheme. In particular, (b) yields an average 7.12% AIR improvement, while (a) yields an average 8.50% AIR improvement. This is because immediate reward collection can be easily affected by noises and does not account for action impact that takes a while to manifest itself. Figure 17 illustrates the comparison in one experiment.

We expected that the incremental scheme (c) would perform slightly better than (a), since it could use more information and would not need to wait for the full observation. Contrary to our expectation, in our experiments, the incremental scheme performs slightly worse than the delayed scheme, with a 8.45% AIR improvement. One reason is that, if the environment is relatively steady, adding cost with partial observation does not provide much new information. Additionally, the collected cost from the partial schema might not be distributed evenly across the observation window. In this case, the incremental scheme would be misled by the partial observations. However, the incremental scheme does have the advantage of response faster in case the system is under rapid change, so it can make decisions based on the latest cost data.

8.8 Safe Guards

The safe guards can influence the system in many ways. First, it allows a constant exploration of all action to enable timely adaptation to system changes. In the I/O time out case studies, the bandit could not have readjust to use *UA-LM-RH*, hence losing 3.7% of cost. Second, it prevents early convergence



(a) Session count **(b)** Session duration

Figure 18: Mitigation request handling sessions

to a wrong policy. In the E11 case study, when simulating the bandit without safe guards, we converged (probability > 0.95) to use a single action in 27% of cases, 19% of which was *UA-SR*, the worst action. Third, safe guards decrease the impact of unexpected spikes. In the IO timeout experiment, on 2020-04-25, cascading failures resulted in 106 VM interruptions on a single node for the *NoOp* group. Although this significantly impacted the probability to choose *NoOp*, we still keep exploring that option.

8.9 Scale and Performance

Narya runs in each data center region of Azure. The mitigation engine handles hundreds to thousands of requests daily. The failure predictor processes tens of TBs of health signals per day. Figure 18a shows the number of daily mitigation request sessions (including all fault handling), and the number of requests that go through our A/B testing experiments and requests handled by our Bandit model. Figure 18b shows the CDF of the session duration of the mitigation actions.

9 Discussion and Limitations

Lessons. We share some operational issues and summarize the lessons we learned from running Narya in production.

First, given the sheer complexity of Azure cloud infrastructure, it is inevitable that some Narya decision could go wrong. We encountered two kinds of service misbehavior: (1) some prediction rules are outdated and incorrectly mark many nodes in a short period of time; (2) an increase of customer impact that is not incorporated within the cost model. For (1), the issue would impact AB testing and cause Bandit to take time to adjust. Our rate limit mechanism described in Section 7.2 would help. We also designed a separate anomaly detection algorithm to catch such misbehavior so we can pause and refine the offending prediction rules. To overcome (2), we added monitoring of the support tickets filed by customers.

Second, as Narya consumes telemetry signals from the whole stack, Narya may be broken if the updates of host OS, firmware, and hardware involve uncoordinated schema changes. We recently had an issue in which the schema change of a few critical OS signals was not captured by Narya. Our monitoring component caught the issue and we had to patch Narya. Besides schema changes, the data and label quality may also fluctuate due to improvements or regressions of tracing capability introduced by different component teams.

The aforementioned challenges can be addressed if the cross-team collaboration and communication are perfect, which unfortunately is not realistic in large organizations. Through continuous learning from failures, we build multiple channels based on social alignment principles to include relevant component teams, so that the right team can be involved in time to avoid broken contract, adjust prediction rules, etc.

While we try to ensure sufficient communication, we cannot just rely on it. We build a comprehensive monitoring pipeline that detects anomalies at all layers for Narya, from input data to prediction results, mitigation actions, etc. We proactively investigate alerts and follow up on issues caused by external dependencies or fix Narya's own defects. Moreover, we re-train our ML model on a regular basis to accommodate the evolution of telemetry data and label quality.

Third, Narya may output unexpected decisions, which require verifying its correctness and diagnosing the root cause. In general, diagnosing issues in Narya' Bandit decisions is easy. The exploration model is explainable and solely depends on the total VM reboots observed and the total nodes observed. Any unexpected change in probability can be traced down to the observations that had large customer impact.

Limitations. We describe several limitations of Narya. While Narya can be fully automated, it currently still involves some human intervention to analyze the experiment results and update the system. This is because our cost model for customer impact is incomplete. We believe limited human intervention is key to catch any gaps in customer complaints and improve.

We currently focus on predicting and mitigating hardware or firmware-induced VM failures. We plan to extend Narya to software-induced VM failures. While generic software failure prediction is very challenging due to their frequent changes and complex dependencies, there are potentials for addressing issues like memory leak, repeated crashes, and timeout bugs.

The multi-armed Bandits model we use has the advantage of simplicity and is easy to explain the mitigation decision. However, this model can segment the data. In particular, Narya divides nodes into different experiments based on fault code and node metadata (e.g., h/w generation). But mitigation actions for nodes from different experiments may share the same characteristics, which may not be learned because each model is only trained on the data from its corresponding experiment. We are exploring with contextual Bandit model [23] to include heterogeneous nodes in an experiment and feed context information like node features to the model input.

10 Related Work

Our work is related to three subareas in system resilience: failure detection, prediction and mitigation. Failure detection has been extensively studied, while failure prediction and mitigation are not as well explored. Narya's major contribution is improving the latter two in the context of a large-scale, production cloud VM infrastructure, and designing an end-to-end preventive mitigation service to achieve failure *avoidance*.

Detecting crash failures reliably and quickly in asynchronous distributed systems is a classic topic [3, 5, 6, 9, 13, 22, 34]. Recent work has discussed the prevalence of gray failures [11, 17, 26] in cloud. Panorama [16] proposes to leverage observability to detect gray failures [17]. Narya focuses on predicting failures ahead of time. Many of the failures we target fall into gray failure category. But our aim is to identify risky hosts *before* they cause customer impact.

Several recent work proposes using machine learning to predict disk failures [14, 27, 38] and node faults [25]. Narya predictor aligns with these solutions' basic approach. But we focus on predicting failures in the complex VM host environment as a whole and those that have customer impact. Additionally, we design the prediction pipeline to closely integrate with the mitigation engine.

The Recovery-Oriented-Computing project [31] advocates the importance of failure mitigation, particularly reboot [4]. Piegon [21] proposes to expose uncertainty of failures to allows applications for better failure reactions. But applications have to manually decide whether to wait or start recovery. IASO [30] is a framework for detecting fail-slow issues and supports mitigating slow issues with multiple options such as process restart or VM shutdown. But it relies on customers to manually configure the mitigation option.

NetPilot [37] aims to automate the failure mitigation in a data center network by determining the suspected network devices and mitigating failures based on estimated impact. Narya differs with NetPilot in several ways. First, Narya targets automating the failure mitigation of a system with heterogeneous components and complex stack. In our setting, estimating the impact of an action offline is challenging and often mismatches with production observations. Narya takes an online exploration and learning approach. Second, the mitigation actions available in NetPilot are few and simple like device restart. Narya needs to consider diverse and complex actions. Third, Narya aims to *avoid* failures whereas NetPilot focuses on mitigating failures that have occurred. Lastly, Narya is deployed in production at large scale.

A/B testing experimentation is a common practice to test the effects of UI features using production data (user requests). The idea is simple, but it often yields surprising power [19]. Thus, leading companies conduct thousands of A/B experiments annually. Narya mitigation engine adopts the A/B testing methodology in a novel way to the failure mitigation scenario with several changes. Narya also adopts multi-armed Bandits reinforcement learning [32]. Our contribution is addressing several unique challenges and the system support that make the approach work in a large-scale, production cloud infrastructure to avert real cloud VM interruptions.

11 Conclusion

We investigate an increasingly important topic in fault-tolerant system designs—failure avoidance in the context of cloud infrastructure. Drawing from our experience in operating a large production cloud system, we propose a novel online experimentation and learning approach to tackle this problem. We present Narya, an end-to-end service consisting of failure prediction and smart mitigation. Narya continually evaluates the optimal action in production using A/B testing and Bandit models. Narya has been running in Azure compute infrastructure for 15 months and yields a 26% improvement in reducing VM interruptions compared to previous static strategy.

References

[1] Apache kafka.

[2] S. Agrawal and N. Goyal. Thompson sampling for contextual bandits with linear payoffs. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1220–III–1228, Atlanta, GA, USA, 2013.

[3] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS '09, pages 3–3, Monte Verità, Switzerland, 2009.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, pages 31–44, San Francisco, CA, 2004.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[6] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(5):561–580, May 2002.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[8] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, Shanghai, China, 2017.

[9] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, Feb. 2003.

[10] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[11] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 1–14, Oakland, CA, USA, 2018.

[12] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 8–8, Santa Ana Pueblo, New Mexcio, 2013.

[13] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 175–188, Stevenson, Washington, USA, 2007.

[14] G. Hamerly and C. Elkan. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, page 202–209. Morgan Kaufmann Publishers Inc., 2001.

[15] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 545–557. USENIX Association, Feb. 2020.

[16] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018.

[17] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3146–3154. Curran Associates, Inc., 2017.

[19] R. Kohavi and S. Thomke. The surprising power of online experiments. *Harvard Business Review*, 95(5):74–82, 2017.

[20] J. Lee, Y. Lee, J. Kim, A. Kosiorek, S. Choi, and Y. W. Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019.

[21] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, pages 427–442, Lombard, IL, Apr. 2013.

[22] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, Cascais, Portugal, Oct. 2011.

[23] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 661–670, Raleigh, North Carolina, USA, 2010.

[24] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, and M. Chintalapati. Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, Feburary 2020.

[25] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang. Predicting node failure in cloud service systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 480–490, Lake Buena Vista, FL, USA, 2018.

[26] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20. USENIX, Feburary 2020.

[27] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making disk failure predictions SMARTer! In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 151–167. USENIX Association, Feb. 2020.

[28] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS '17, page 4768–4777, Long Beach, California, USA, 2017.

[29] J. C. Mogul and J. Wilkes. Nines are not enough: Meaningful metrics for clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 136–141, Bertinoro, Italy, 2019.

[30] B. Panda, D. Srinivasan, H. Ke, K. Gupta, V. Khot, and H. S. Gunawi. IASO: A fail-slow detection and mitigation framework for distributed storage services. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 47–61, Renton, WA, USA, 2019.

[31] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, EECS Department, University of California, Berkeley, Mar 2002.

[32] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[33] W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 12 1933.

[34] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 55–70, The Lake District, United Kingdom, 1998.

[35] X. Wang, L. Yu, K. Ren, G. Tao, W. Zhang, Y. Yu, and J. Wang. Dynamic attention deep model for article recommendation by learning human editors' demonstration. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 2051–2059, Halifax, NS, Canada, 2017.

[36] B. L. Welch. The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[37] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating datacenter network failure mitigation. *SIGCOMM Comput. Commun. Rev.*, 42(4):419–430, Aug. 2012.

[38] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou, M. Chintalapati, and D. Zhang. Improving service availability of cloud systems by predicting disk error. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 481–493, Boston, MA, USA, 2018.