Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Cloud-Scale Infrastructure

Ze Li[†], Qian Cheng[†], Ken Hsieh[†], Yingnong Dang[†], Peng Huang[*], Pankaj Singh[†]
Xinsheng Yang[†], Qingwei Lin[‡], Youjiang Wu[†], Sebastien Levy[†], Murali Chintalapati[†]

[†]*Microsoft Azure* [*]*Johns Hopkins University* [‡]*Microsoft Research*

Abstract

Modern cloud systems have a vast number of components that continuously undergo updates. Deploying these frequent updates quickly without breaking the system is challenging. In this paper, we present Gandalf, an end-to-end analytics service for safe deployment in a large-scale system infrastructure. Gandalf enables rapid and robust impact assessment of software rollouts to catch bad rollouts before they cause widespread outages. Gandalf monitors and analyzes various fault signals. It will correlate each signal against all the ongoing rollouts using a spatial and temporal correlation algorithm. The core decision logic of Gandalf includes an ensemble ranking algorithm that determines which rollout may have caused the fault signals, and a binary classifier that assesses the impact of the fault signals. The analysis result will decide whether a rollout is safe to proceed or should be stopped.

By using a lambda architecture, Gandalf provides both real-time and long-term deployment monitoring with automated decisions and notifications. Gandalf has been running in production in Microsoft Azure for more than 18 months, serving both data-plane and control-plane components. It achieves 92.4% precision and 100% recall (no high-impact service outages in Azure Compute were caused by bad rollouts) for data-plane rollouts. For control-plane rollouts, Gandalf achieves 94.9% precision and 99.8% recall.

1 Introduction

In a cloud-scale system infrastructure like Microsoft Azure, various teams need to frequently make software changes in code and configurations to deploy new features, fix existing bugs, tune performance, *etc.* With the sheer scale and complexity of such infrastructure, even a small defect in updating one component may lead to widespread failures with significant customer impact such as unavailability of the virtual machine service. Indeed, many catastrophic service outages are caused by some small changes [2, 3, 4, 5, 19].

Each software change, therefore, must be rigorously reviewed and extensively tested. Nevertheless, some bugs could remain uncaught due to the discrepancies between testing and production environment in cluster size, hardware SKU (stock keeping unit), OS/library versions, unpredictable workloads, complex component interactions, *etc.*

Thus, even when a software change passes testing, instead of updating all nodes at once, it is common practice to apply the change to production gradually following a safe deployment policy in the order of stage, canary, pilot, light region, heavier region, half region pairs, other half of region pairs. Figures 1 and 2 respectively show the scope and duration for rollouts in Azure infrastructure. More than 70% of the rollouts target multiple clusters, and more than 20% of the rollouts last for 1,000+ minutes.

Such characteristics imply that the very process of rolling out changes in production during the deployment phase presents an opportunity to catch bad changes in a realistic setting. If faults caused by a deployment can be caught at an early stage, it allows the release manager to stop the bad deployment and roll back the change in time to prevent it from causing broader impact such as a whole-region or worldwide unavailability.

Yet, accurately assessing the impact of a deployment in a cloud system is challenging. Solutions like component-level watchdogs [31] that check for a handful of component-level fault signatures are effective for capturing obvious, immediate issues. They alone, however, are insufficient in catching production issues that are minor locally but severe globally across clusters and/or regions. They may also miss latent issues such as memory leaks that happen hours after a rollout. Additionally, a component-level watchdog may fail to catch issues that arise only when interacting with other components, *e.g.*, cross-components API contract violations.

Besides false negative, false alarm poses another challenge. Figure 3 plots the number of deployments in Azure during a recent three-month window. We can see that hundreds of rollout tasks are happening every day. In addition, transient faults such as service API timeouts and temporary network issues are common in production. All of these events can eas-

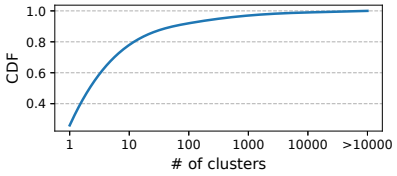

Figure 1: CDF of the scope (number of target clusters) of a rollout.

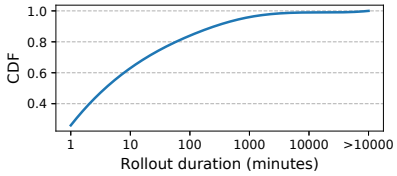

Figure 2: CDF of the rollout duration (in minutes).

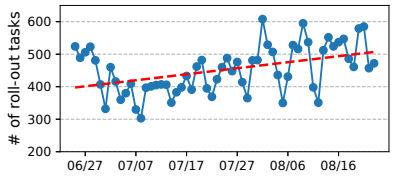

Figure 3: Number of rollout tasks per day. The red dotted line is a trend line.

ily mislead a local deployment health monitor to incorrectly attribute a failure to an innocent rollout. These false alarms would cause the innocent rollout to be stopped and prevent timely changes from being applied. Even worse, developers also waste significant time and resources in investigating such false alarms. Consequently, they would not trust future decisions from the monitoring system.

In this paper, we present Gandalf, an end-to-end analytics service that addresses the aforementioned challenges to ensure safe deployment in cloud infrastructure. Instead of analyzing each rollout separately based on individual component logs, Gandalf takes a top-down approach to assess the impact of rollouts holistically. Gandalf continuously monitors a rich set of signals from the infrastructure telemetry data including service-level logs, performance counters, and process-level events. When a system anomaly is detected, Gandalf analyzes if it is caused by a rollout. If a bad rollout is identified, Gandalf makes a "no-go" decision to stop it. Gandalf also provides detailed supporting evidence and an interactive front-end for engineers to understand the issue and the root cause easily.

The core decision logic of Gandalf is a novel model composed of anomaly detection, correlation analysis and impact assessment. The model first detects anomaly from raw telemetry data. It then identifies if a rollout is highly correlated to the detected failures through both temporal and spatial correlation and an ensemble ranking algorithm. Finally, the model uses a Gaussian discriminant classifier to decide if the impact caused by the suspicious rollout is significant enough to stop the deployment.

We design Gandalf system with a lambda architecture [6], combining a real-time decision engine with a batch decision engine. The real-time engine monitors a one-hour time-window before and after the deployment to detect immediate issues; the batch engine analyzes system behavior in a longer time-window (30 days) to detect more complex, latent issues. When Gandalf identifies a bad rollout, it automatically notifies the deployment engine to stop the rollout and fires a ticket with supporting evidence to the owning team.

Gandalf has been running in production for more than 18 months to ensure the safe deployment of Microsoft Azure infrastructure components (*e.g.*, host agents for Compute and Network, host OS). In an 8-month usage window, for data-plane rollouts, Gandalf captured 155 critical failures at the early stage and achieved a precision of 92.4% with 100%

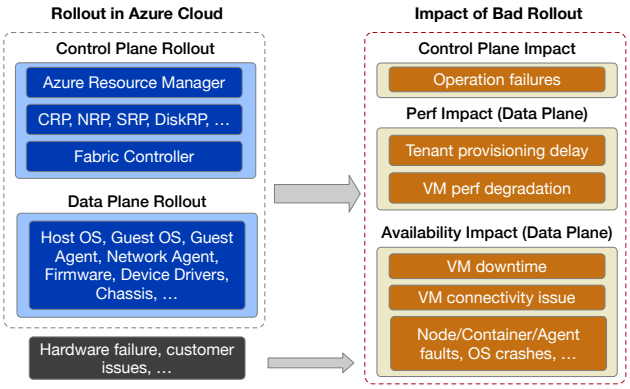

Figure 4: Different rollouts in Azure and impact of a bad rollout.

recall (meaning that no high-impact incidents, *i.e.*, Sev0-2 outages, were caused by bad rollouts); for control plane, Gandalf achieved 94.9% precision and 99.8% recall, with only two missed issues and two false alarms while monitoring 1200+ region-level deployments. Gandalf has made a significant contribution to getting Azure availability closer to its 99.999% objective by limiting the blast radius of customer VM downtime caused by unsafe rollouts.

Gandalf has also improved the deployment experience for release managers: (1) from looking at scattered evidence to using Gandalf as a single source of truth; (2) from being skeptical about Gandalf decisions to enforcing them; (3) from ad-hoc diagnosis to interactive troubleshooting.

2 Background and Problem Statement

2.1 Deployment in IaaS Cloud

In IaaS cloud, the software stack on the physical nodes and virtual machines (VM) consists of many layers of components. Each component may frequently undergo changes on its independent rollout schedule to add features and fix bugs. These rollouts need to be executed with high velocity and minimum customer impact. For example, in early 2018, Azure quickly deployed a fix to mitigate the Meltdown [30] and Spectre [26] CPU vulnerabilities through a host OS update to keep Azure customers secure. The updates were deployed to millions of nodes that host customer VMs.

As shown in Figure 4, two main kinds of rollouts happen in Azure. The *data-plane* rollouts deploy changes for com-

ponents running within the hosting environment of customer VMs. For Azure, those components include the host OS, the guest OS, and various software plugins, called agents, inside the host and guest OS. In contrast, *control-plane* rollouts deploy changes to tenant-level services. These services are composed of distributed running instances to manage the system infrastructure and provide interfaces for their functionalities. These control-plane components include the Azure Resource Manager (ARM) [1], which allows customers to query, create, update, and delete VMs with REST APIs and the CRP/NRP/DiskRP (Compute/Network/Disk Resource Provider), which handle customer requests and provision corresponding resources (e.g., virtual disks for VMs). These services are typically structured as loosely-coupled microservices in Azure that communicate with each other via APIs. While loose coupling allows each service to be deployed independently, their deployments also have intricate impacts on each other. Thus, a simple change in one service, while causing no failure in that service, might break the contract with another service and affect a large number of customer API calls if the defective change gets deployed to production.

2.2 Deployment Monitoring System

Requirements To ensure high availability of the VMs and services, rollouts are carefully monitored. Traditionally, safe deployment is a manual process that relies on email communications, multi-party approval, ad-hoc build test validation, and experience-based decisions, which is unsustainable at the scale of Azure. An automated deployment monitoring system is needed to globally oversee rollout progress and automatically stop bad rollout before it causes widespread impact. A deployment monitoring system should detect various anomalies in the large volume of infrastructure telemetry data. In addition, the monitoring system should accurately analyze if the observed failure is caused by a bad deployment or by another issue (e.g., random hardware faults). For the former case, the system should attribute the failure to the responsible deployment among all the ongoing deployments, stop the rollout immediately, and provide supporting evidence for developers to speed up further investigation. For the latter case, the system should not incorrectly blame and stop an innocent deployment.

Target Based on our experience, four kinds of failures happen in production environment: *(1)* hardware issues that happen randomly, *e.g.*, due to firmware bugs, temperature; *(2)* chronic software "hiccups", *e.g.*, due to race conditions; *(3)* hardware-induced outages, *e.g.*, power outage, broken network cable; *(4)* software outage due to bad code or settings in a recent build. Ambient software/hardware issues can be surfaced through separate anomaly detection solutions. In this work, we focus on deployment-related outages, *i.e.*, category *(4)*. Among the four layers of safe-deployment mechanisms

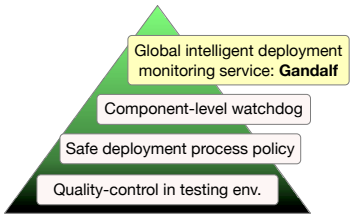

Figure 5: Azure's four-layer mechanisms to ensure safe deployment.

in Azure as shown in Figure 5, Gandalf serves as the last safeguard, focus on catching system-level failures, including non-obvious and latent issues, caused by bad deployments.

3 Gandalf System Design

The importance of catching failures during rollouts and the complexity of rollouts in Azure infrastructure motivate the design and implementation of *Gandalf*. Gandalf is an end-to-end, continuous monitoring system for safe deployment. It automates the assessment of rollout impact, the approval or stopping of a rollout, the notification to the owning teams, and the collection of detailed evidence for investigation.

3.1 Design Challenges

In designing Gandalf, we need to address several challenges.

Supporting changes in system and signals. In Azure infrastructure, hundreds of thousands of update events with a large number of fault signals happen every day across the software stack in millions of physical nodes, VMs, and tenants. Gandalf needs to ingest a comprehensive set of data sources and efficiently process them in order to provide timely responses to developers. Furthermore, since Azure extensively employs agile development and micro-service designs, new components emerge and existing components evolve with changing failure patterns and telemetry signals [37]. Gandalf needs to support easy onboarding of new components and telemetry signals while maintaining a robust core decision logic.

Dealing with ambient noise. Ambient faults happen frequently due to diverse reasons such as hardware faults [20], network timeouts, and gray failures [24]. Many of them are unrelated to deployments and can be successfully tolerated. Gandalf needs to deal with such noise. Figure 6 shows an ambient noise example: container faults happened before and after a host OS update on 200+ clusters in Azure; but they were not caused by the host OS deployment—instead, they were caused by firmware defects that prevent the container from starting. User behaviors also affect failure patterns. For example, a customer might invoke a large number of `CreateVM` calls during weekdays. This could lead to an increase of API failures on weekdays and a decrease on weekends. Only monitoring the increase of faults after a deployment could then result in a wrong conclusion.

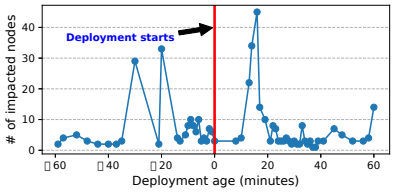

Figure 6: Ambient faults in deployment.

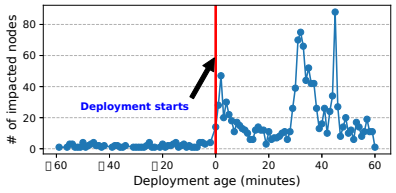

Figure 7: Spike of faults after deployment.

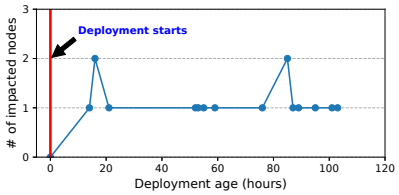

Figure 8: Latent faults after deployment.

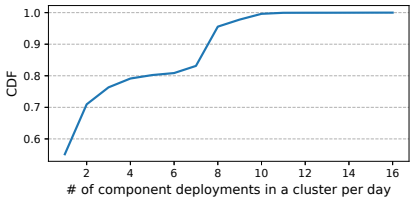

Figure 9: CDF of component count rolled out per cluster in a day.

Balancing speed and coverage. Developers want to get immediate feedback on bad rollouts. Figure 7 shows a typical spike of node faults that happened minutes after a bad rollout. Gandalf needs to quickly alert and stop the bad deployment upon detecting such failures. But a quick decision may be unsound if the failure patterns are subtle. Figure 8 shows an example of a latent issue detected more than 32 hours after the deployment. From the figure, we can see that there is no obvious spike after the deployment of the component. The faults are observed slowly in a long period. Those latent failures are usually only triggered by specific user workloads, e.g., accessing a specific directory or turning on a system service. Therefore, Gandalf needs to detect cross-cluster, latent issues even when a rollout has been ongoing for a while. Covering such latent issues takes longer time by nature.

Identifying the culprit. An *n*-to-*n* mapping relationship exists between components and failures: one component may cause multiple types of failure, while a single type of failure may be caused by issues in multiple components. Figuring out which failure is likely caused by which component is not easy due to the complexity of component behaviors. Figure 3 shows that more than 300 deployments take place in Azure every day and the number is increasing. Figure 9 further plots the number of deployment on a cluster per day, showing that about 45% of the clusters have multiple deployments per day.

3.2 System Overview

Figure 10 shows the overview of Gandalf. Gandalf takes a top-down approach in deployment monitoring by consuming telemetry data across clusters for holistic analyses. Gandalf processes three types of data: (1) *performance data* such as CPU performance counters and memory usage in the node; (2) *failure signals* such as agent faults, container faults, OS crashes, node reboots and API call exceptions; (3) *update events* that describe the component deployment information.

In order to balance speed and coverage for safe deployment, the analysis engine of Gandalf is structured in a lambda architecture [6] with a speed layer and a batch layer. The speed layer focuses detects simple, immediate issues and provides quick feedback to developers. The batch layer detects latent, more complex failures and provides more detailed evidence for developers to investigate the issue.

The analysis results from the streaming processed data and batch processed data are consolidated into the serving layer. The serving layer is built as a highly-reliable and scalable web service. The web service stores the analytics results in batch and streaming tables and provides interfaces for various reporting applications to consume the results. The applications include a monitoring front-end for developers to view the rollout status in real-time, a diagnosis UI for investigating problematic rollout, and REST APIs to directly query the result data. Based on the decisions, Gandalf will notify the corresponding component team and create an incident ticket accordingly. Besides notifications, Gandalf publishes the binary decisions for different components into a key/value store to communicate with the deployment engine. The deployment engine subscribes to the signals in the key/value store and stops the rollout if a “no-go” decision is made.

3.3 Data Sources

As a deployment monitoring system, Gandalf continuously ingests deployment events in the Azure infrastructure. These events describe the software build version along with the deployment timestamp and location information. To focus on the system-level impact of rollouts, Gandalf consumes comprehensive signals from various data sources such as service logs, Windows OS events, performance counters, and machine/process/service-level exceptions. Typically each signal is ingested from a separate table in the telemetry data collected in Azure infrastructure. For certain signals, Gandalf performs pre-processing to parse the raw data (e.g., log messages) and extract a failure signature (e.g., an error code). These pre-processed signals are aggregated based on their timestamps, node IDs and service types during the analysis.

When onboarding a new component to Gandalf, the component team needs to provide information about where Gandalf can get the deployment events and the telemetry signals relevant to the component. To ease the correlation analysis

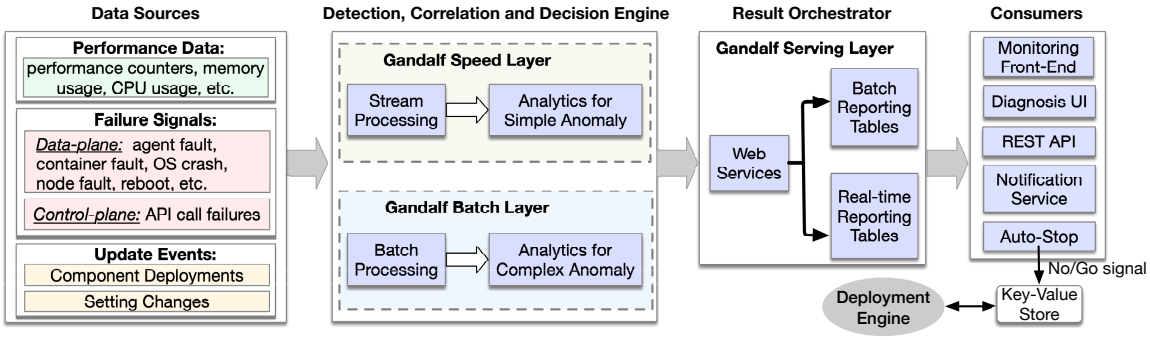

Figure 10: Overview of Gandalf system.

Attribute	Description
Timestamp	When the deployment event completes
Location	Node, cluster, region, etc.
Pivot Group	Hardware SKU, environment, etc.
BuildVersion	Build version identifier
AdditionalInfo	Additional information of the event

Table 1: Gandalf deployment event input data schema.

Attribute	Description
Timestamp	When the fault occurs
Location	Node, cluster, region, etc.
Pivot Group	Hardware SKU, environment, etc.
Signature	Fault signature
AdditionalInfo	Additional diagnosis information

Table 2: Gandalf fault signal input data schema.

between telemetry signals and deployment events, Gandalf requires the information to be structured in a unified data schema as shown in Table 1 and Table 2.

3.4 Stream and Batch Processing

To balance speed and coverage, Gandalf is designed in a lambda architecture [6] with both streaming and batch analysis engines. The speed layer consumes data from a fast pipeline, Microsoft Kusto [7], which is a column-oriented cloud storage supporting analytics with a few minutes of data source delay and up to seconds of query delay. Kusto has a custom query language based on the data-flow model, with native support for streaming operators. Although Kusto has short delays, it cannot efficiently handle a large volume of data using complex algorithms. Therefore, the analysis engine in the speed layer only considers fault signals that happen 1 hour before and after each deployment in each node, and runs lightweight analysis algorithms to provide a rapid response. In Azure, most catastrophic issues happen within 1 hour after the rollout. Latent faults occurring after 1 hour will be captured by the batch layer later.

The Gandalf batch layer consumes data from Cosmos, which is a Hadoop like file system that supports SQL-like query language with up to hours of data source delay. Despite the relatively long delay, Cosmos is an ideal platform for processing an extremely large volume of data using complex models (*e.g.*, it supports external C++ plugins) to detect complex failure and latent issues. This allows the batch layer to analyze faults in a larger time window (30-day period) with advanced algorithms. The lambda architecture allows us to provide both fast decision making and higher coverage over time. A no-go decision can be triggered anytime within the window if the impact scope is large enough.

Both stream and batch analysis in Gandalf are performed in an incremental fashion. Every 5 minutes, the stream processing fetches the latest data streams from Kusto and passes it to the analysis engine in the speed layer. The batch processing runs as an hourly Cosmos job to process the data since the last processing time. The partial results from analyzing each 5 mins mini-batch or hourly batch are aggregated with other partial results in the analysis window to update the overall result. The incremental analysis improves the efficiency as well as the fault tolerance of Gandalf—if the analysis job is restarted, it can resume from the last checkpoint.

3.5 Result Orchestration and Actions

The serving layer of Gandalf is implemented as a highly-reliable and scalable web service using the Azure service fabric framework [8]. After each run of Gandalf's anomaly detection and correlation algorithms (which we will describe in Section 4), the results from the speed and batch layers are stored in two separate reporting tables through the web service. These results contain the deployment impact assessment, the recommended decisions (*"go"* or *"no-go"*), the anomaly patterns, the correlation information, etc.

In general, the telemetry data Gandalf analyzes is both streamed into Kusto and dumped into Cosmos hourly/daily. Given that the data ingested by the speed and batch layer is essentially the same, the reporting results in the two tables are mostly consistent. For example, if the speed layer quickly detects a bad deployment, the batch layer will likely catch it as well, albeit slower. The scenarios when they are inconsistent is mainly when the batch layer reaches a *"no-go"* decision

while the speed layer decides a *go*. This is by design since the batch layer makes more informed decisions and covers latent issues that the speed layer cannot detect.

Various DevOps applications pull the results from the reporting tables. This way, the Gandalf system is well integrated into the DevOps workflow. Among the applications, the most important one is the notification service. When the notification service notices a new no-go decision, it sends an email about the decision to the owning team and creates an incident ticket with details. It also notifies the deployment engine via a key-value store to approve or stop the rollout.

3.6 Monitoring and Diagnosis Front-End

Gandalf provides a web front end to enable real-time rollout monitoring and issue diagnosis support for release managers and developers. The feature teams can proactively watch the rollout KPIs (e.g., rollout progress, NodeFaults, Container Faults, OS Crashes, Allocation Failures and etc.) in real-time while waiting for the decision from the Gandalf notification service. After the decision of a rollout is made and sent to the corresponding team, Gandalf provides information to help developers investigate the issue and make a quick fix as needed. For example, Gandalf provides pivot information of the identified issues (*i.e.*, the issue happens only in instances that have specific attributes like SKU).

The Gandalf front-end provides the following views: (i) a binary decision page that summarizes all rollout decisions for different components and build versions in different environments; (ii) a rollout profile page that displays the batch processed decisions and associated diagnosis information; (iii) an issue profile page for a bad rollout with diagnosis information such as the impacted nodes and clusters or the trend in different environments; (iv) a real-time tracking page that shows the rollout progress, related failures, etc.

4 Gandalf Algorithm Design

Existing Algorithms. In designing the algorithms for Gandalf, we considered existing options from supervised learning, anomaly detection and correlation analysis but found major limitations for each of them. Supervised learning is difficult to apply because system behaviors and customer workloads as well as failure patterns and failure-update correlation keep changing. In addition, learning from historical change behaviors does not necessarily help predict the future mapping between failure patterns and new updates. Existing anomaly detection algorithms alone are also insufficient. This is because many rollouts may happen simultaneously in the infrastructure but anomaly detection itself does not tell which deployment is responsible for the anomalies. For correlation analysis, most state-of-the-art methods focus on temporal correlation based on Pearson correlation [34], which cannot capture the complex causal relationship in our scenario.

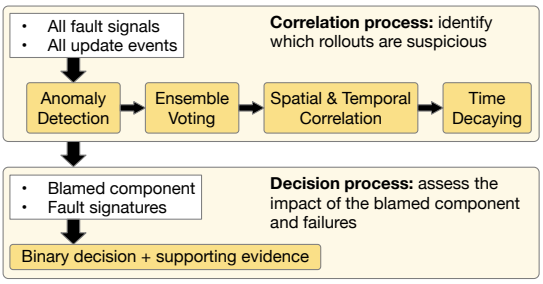

Figure 11: Gandalf correlation model.

Overview. The Gandalf model consists of three main steps: *(1)* anomaly detection detects system-level failures from raw telemetry data; *(2)* correlation analysis identifies the components responsible for the detected failures among multiple rollouts; *(3)* the decision step evaluates the impacted scope and decides whether the rollout should be stopped or not. The correlation step in (2) is further divided into four parts, namely, ensemble voting, temporal correlation, spatial correlation and exponential time decay. Figure 11 shows the overall analysis process in the Gandalf. In the following Section, we describe each step of the algorithm in detail.

4.1 Anomaly Detection

The raw telemetry data that Gandalf analyzes, such as OS events, log messages, and API call statuses, may be imprecise. Therefore, Gandalf first derives concise fault signatures from raw data to distinguish different faults. A raw fault event log usually contains both error codes and error messages. One error code could map to multiple faults if it is too generic. For example, a POST API call that requests compute resources could return `HTTP ERROR 500` for different reasons like `AllocationFailure` or `NetworkExecutionError`. Directly analyzing the error codes would mix different faults, diluting the signal and leading to wrong conclusions. The error messages, on the other hand, are usually non-structured plain text with many unnecessary details. Gandalf processes the raw error messages and applies text clustering [10] to generate fault signatures. We first replace the unique identifiers such as VM ID, subscription ID with dummy identifiers using an empirical log parser similar to prior work [18, 22]. Then we run a simplified incremental hierarchical clustering model [36] to group up all processed text into a set of error patterns, *e.g.*, "Null References" grouped together with `NullReferenceException`.

After obtaining the fault signatures, Gandalf detects anomalies based on the occurrences of each fault signature. Ambient faults, such as hardware and network glitches or gray failures [24], are common in a large-scale cloud system. Simple threshold-based anomaly detection is ineffective because the system and the customer behaviors change over time. With thousands of signatures, it is also unrealistic to manually set thresholds for each. Gandalf instead estimates the baseline from past data using Holt-Winters forecasting [14] to detect

anomalies. The training period is set to the past 30 days and the step interval is set to one hour. When the observed value deviates from the expected value by more than 4σ, the point will be marked as an anomaly.

For some components, the occurrences of different fault signatures vary significantly. For example, the volume of client errors could be much higher than platform errors. To better compare the impact of different fault signatures, Gandalf calculates z-score [28], $z_i = \frac{x_i - \mu}{\sigma}$, for each anomaly against historical data in its correlation process.

4.2 Correlation Analysis

A detected failure may not be caused by a bad rollout but other factors such as random firmware issues. In addition, at any point, many concurrent rollout tasks can take place in a large system. Therefore, once anomalies are detected, Gandalf needs to correlate the observed failures with deployment events, and evaluate the impact of the failure on the fly. Note that this identified component might be the triggering component but not necessarily always the root cause.

4.2.1 Ensemble Voting

Since many components are deployed concurrently, we use a vote-veto mechanism to establish the relationship between the faults and the rollout components. For a fault e that happens at timestamp t^f and a rollout component c deployed at t^d on the same node, each fault e votes for all the components deployed before it (i.e., $t^d < t^f$) within a window size w_b and vetoes all the components deployed after it (i.e, $t^d > t^f$) within a window size w_a. Since the deployments are rolled out continuously on different nodes at different time, as shown in Figure 12, we aligned the votes $V(e,c)$ and vetoes $VO(e,c)$ for deployed component c across all nodes based on the fault age as defined as $age(e,c)=t^f - t^d$. P_i are the votes aggregated on WD_i as

$$P_i = \sum_k V(e,c|WD_i), \tag{1}$$

where $age(e,c) < WD_i$. By default, 4 different hour windows are used as WD_1=1, WD_2=24, WD_3=72, WD_4 is the duration between the deployment and the latest data point; k is the number of nodes with the pairs. Similarly, B is the veto aggregated in a single 72 hour window as

$$B = \sum_k VO(e,c|WD_{-1}), \tag{2}$$

where $age(e,c) < WD_{-1}$ and WD_{-1}=72.

4.2.2 Temporal and Spatial Correlation

After ensembling the votes of faults to the components, we calculate the temporal correlation score as

$$ST(e,c) = \sum_{i \in [1,4]} w_i \log\left(\frac{(P_i - B + 1)}{B+1}\right), \tag{3}$$

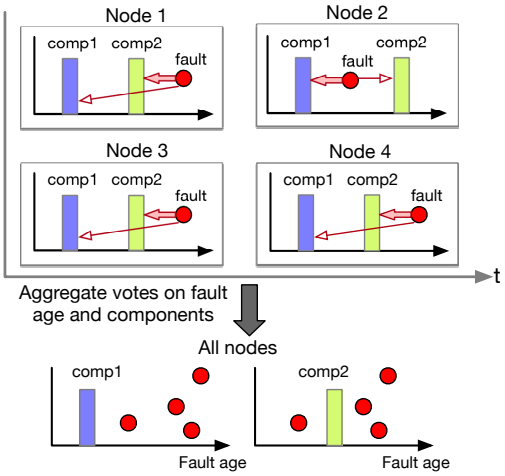

Figure 12: Faults alignment by fault age during ensemble voting. The circle represents a fault. The vertical bar is when a component gets deployed in a node. The arrow blames a fault to a deployment, where the arrow size represents correlation strength.

where $P_i > B$. w_i is the weight of the time window WD_i in Section 4.2.1. This kernel function tries to filter out the ambient faults.

As we do not have ground-truth samples to train the values of w_i in the above equation, we set them empirically. A naïve way would be to assign them the same value. This does not work well in practice because a component deployed closer to a fault usually has a higher correlation, implying the constraint $w_1 > w_2 > w_3 > w_4$. We learned over time that setting *exponential weights* (EW) for w_1 to w_4 works well. The intuition behind exponential weights (EW) is that faults happening right after the deployment are much more likely to have causal relationship than the faults happening a long time after the deployment.

We then evaluate the spatial correlation through

$$SS(e,c|t_1,t_2) = N_f / N_{df}, \tag{4}$$

where N_f is the number of nodes with fault e during the deployment period t_1 to t_2 for component c, and N_{df} represents the total number of nodes with fault e, regardless of whether c was deployed during the same period. If $SS(e,c|t_1,t_2) < \beta$, where β is the confidence level, we will ignore the blaming of the pair. The confidence level can be set as 99% or 90% for different sensitivity. We then identify the blamed component c_j by associating the faults to the component with the largest temporal correlation:

$$blame(e) = \arg\max_{c_j} ST(e,c_j) \tag{5}$$

4.2.3 Time Decaying

The blaming score is calculated based on the fault age as shown in Equation 3. If the same fault signature appears again, the fault may still blame the old rollout if the fault age between

the old rollout and the fault signature is smaller. We need to focus on new rollouts and gradually dampen the impact of the old rollout because newly observed faults are less likely to be triggered by the old rollout. In order to achieve this, we apply an exponential time decay factors on the blaming score:

$$blame(e) = blame(e) * (\frac{e^{-t} - e^{-w_s}}{e^{-1} - e^{-w_s}}) * b + a \qquad (6)$$

4.3 Decision process

Finally, we make a go/no-go decision for the component c_j by evaluating the impacting scopes of the deployment such as the number of impacted clusters, the number of impacted nodes, number of customers are impacted, *etc.* Instead of setting static thresholds for each feature, the decision criteria are trained dynamically with a Gaussian discriminant classifier [9]. The training data is generated from historical deployment cases with feedback from components teams. Note that the impacting scope feature set used in this step is typically organizational policy oriented so it is stable and independent from software changes or bug fixes. Thus it is feasible to obtain good labels for this learning approach comparing to the input features used in correlation analysis.

4.4 Incorporating Domain Knowledge

Gandalf by default treats the input fault signals equally but also allows developers to specify the importance of certain faults with customizable weights. The weights are relative values ranging from 0 to 100, representing the least to the most importance. The default weight for a fault signal is 1.

Weights are usually adjusted by developers reactively, e.g., after investigating a reported issue. For example, developers for certain service may find out the `TimeoutException` tend to be noisy so they reduce its weight to 0.01 for that service; for the Disk Resource Provider, developers may set the weight of `NullReferenceException` to 10 so that Gandalf becomes more sensitive to this failure signature because it is a strong indicator of a code bug. If the weight is set to 0, this failure signature is whitelisted. For example, since developers know that during the rollout of NodeOSBaseImage, node reboots are expected, the weight for `NodeReboot` can be set to 0 to exclude it from the correlation analysis. In general, developers rarely set a fault weight to 0 to avoid missing true issues unless the signal keeps causing false alarms or to encode special rule like the previous example. Since Gandalf exposes the weight settings to developers through a database table, developers sometimes use scripts to adjust weights in a batch, e.g., lowering weights for all `AllocationFailure*` signatures.

5 Evaluation

Gandalf is a production service in Azure. In this Section, we evaluate its business impact, provide three case studies, and analyze the effectiveness of the core algorithms (Section 4).

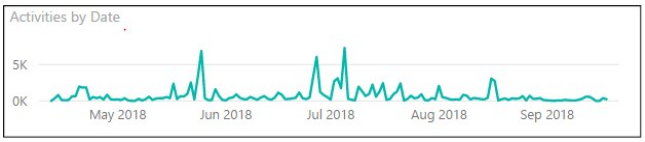

Figure 13: User activities of Gandalf real-time monitoring UI.

5.1 Business Impact

Adoption. The Gandalf service has been running in production and monitoring the Azure infra rollout safety for more than 18 months. It has been widely adopted for the deployments of data-plane components and control-plane services (Section 2.1) across the entire fleet. Specifically, Gandalf currently monitors 19 component rollouts in the data plane including Host OS updates, Agent Package updates, GuestOS updates, *etc.*, as well as 4 control-plane component updates including Compute Resource Provider, Disk Resource Provider, Azure Front End, and Fabric Controller. Figure 13 shows the usage of Gandalf front-end by release managers and developers. Usually, hundreds of page-visits occur every day. When a large rollout happens, the daily number of page visits can reach several thousand.

Scale. The Gandalf system processes on average 270K platform events daily, 770K events on peak days, and logs about 600 million API calls per day in the control plane, including more than 2,000 fault types. The total data volume analyzed is more than 20TB per day.

Deployment Speed. Gandalf has significantly improved the release velocity. For each deployment, Gandalf can make decisions in about 5 minutes end-to-end on the speed layer, and in about 3 hours on the batch layer. Gandalf cuts the deployment time for the entire production fleet by more than half (Figure 14). As a result, billions of customer API requests will benefit from new features much earlier.

Gandalf streamlines the traditionally cumbersome deployment workflow. Prior to Gandalf, the component-level watchdogs were sometimes noisy or missed important failures. Thus, extensive email communications were needed among the release manager, feature owners and dependent component teams to clear suspicious failure alerts and obtain approval. As a 24/7 monitoring system, Gandalf removes the majority of these costs. Meanwhile, Gandalf provides rich supporting evidence for each alert to facilitate further investigation.

As Azure grows with ever more data to analyze, Gandalf can benefit from additional innovations that improve the performance of analytics systems [29, 35, 38]. For example, if we can cut the delay of our streaming layer from 5 minutes to 10s, it will greatly improve the deployment speed. On the other hand, even with ultra-low data processing latency, Gandalf still needs to wait to accumulate enough evidence for high-confidence decisions. If the rollout quality can be predicted beforehand, the system will have higher business impact. We leave this as our future work.

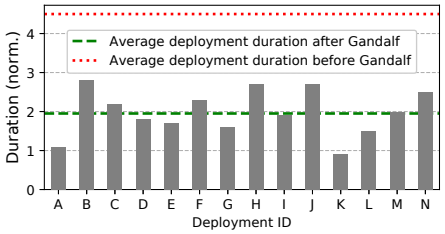

Figure 14: Deployment duration before and after adopting Gandalf

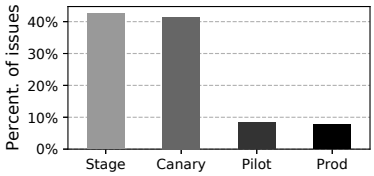

Figure 15: Percentage of issues detected in each environment.

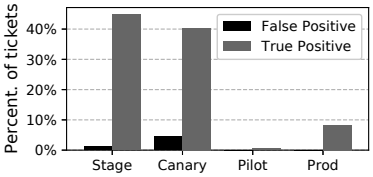

Figure 16: Accuracy of tickets issued by Gandalf in each environment.

5.2 Accurately Preventing Bad Rollouts

Azure enforces a safe deployment policy for all rollouts. Before a component update can be pushed to production, it must pass tests in several environments in the order of Stage, Canary and Pilot. Figure 15 shows the percentage of issues we detected in different environments for rollouts in the data plane. We can see that almost all (99.2%) suspicious rollouts are blocked before reaching production. The rest (0.8%) is blocked in the early stage of production. This means Gandalf effectively limits the blast radius of most bad rollouts.

Once Gandalf stops a rollout, it will send an alert ticket to the corresponding team. Figure 16 shows the accuracy of the alerting in different environments. We can see that most of the false alerting are issued in Stage and Canary environment, which is well aligned to the Gandalf design goal. The false positive rate in Stage and Canary is higher compared to other environments because there are higher levels of noisy failure signals in these environments. Latent issues may mislead the Gandalf service. For example, a faulty agent update accidentally deletes an important folder but does not cause immediate faults. Later, a GuestOS update by customer touched that folder and triggered faults, causing Gandalf to mis-blame the GuestOS deployment.

Overall, in an 8-month usage window from Jan. 2018 to Nov. 2018, Gandalf captured 155 critical failures at the early stage of the data-plane rollouts and achieved a precision of 92.4% with 100% recall (no high-impact incidents were caused by bad rollouts). The detected failures are diverse, including agent faults, OS crashes, node faults, unhealthy containers and VM reboots, which would have caused widespread availability outages. For the control plane, Gandalf has made decisions for 1200+ region-level deployments. The precision is 94.9% and recall is 99.8%. Gandalf filed 39 incidents and only 2 of them are false alarms. Meanwhile, Gandalf automatically approved all other region-level deployments and only missed 2 true issues. One false negative occurred because of incomplete logs. The other was due to the faulty component throwing generic timeout exceptions instead of specific errors, which misled Gandalf.

The most common issues Gandalf caught are compatibility issues and contract breaking issues. Compatibility issues arise when updates are tested in an environment with latest

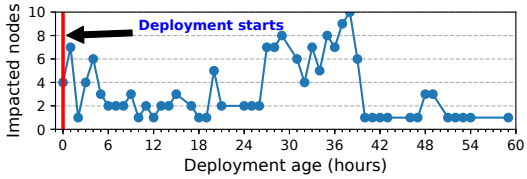

Figure 17: Latent faults caused by network agent deployment.

hardware or software stack but the deployed nodes may have different hardware SKUs or OS or library versions. Contract breaking issues occur when the component does not obey its API specifications and break dependent components.

5.3 Case Studies

We share three representative cases of bad rollouts that Gandalf successfully prevented.

***Case I*: Cross-component Impact.** Release managers tend to ignore faults from other components, which may miss cross-component reliability issues. Gandalf makes informed decision based on anomaly detection and correlation analysis, and is sensitive to such issues. In one case of deploying a Compute Resource Provider (CRP) service update, Fabric Controller (FC) lease failures occurred. When Gandalf first made a no-go decision for the deployment in Canary regions, CRP team claimed these failures were irrelevant to CRP rollout based on their past experience. Therefore, the release manager directly requested to bypass the no-go decision and unblock the rollout. Later on, Gandalf issued another no-go decision in Pilot regions for the same reason, indicating a strong correlation between this failure and CRP rollout. With such evidence, CRP team did a deeper investigation and confirmed it was indeed a regression in CRP. When the customers in Pilot regions reported relevant issues, a hotfix had already been deployed. Because of Gandalf, the regression was caught before it could enter production.

***Case II*: Impact in Specific Region.** After a rollout passes the Pilot regions, release managers typically assume the software updates are in high quality. But Gandalf keeps monitoring throughout the deployment process. Issues that arise at this stage are usually not caused by code bugs of the deployed component but rather the incompatible settings in a specific

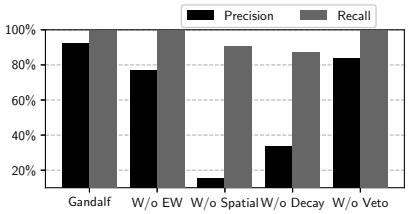

Figure 18: The effectiveness of each Gandalf correlation algorithm.

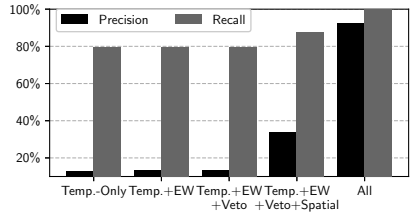

Figure 19: Accumulative effects of correlation algorithms.

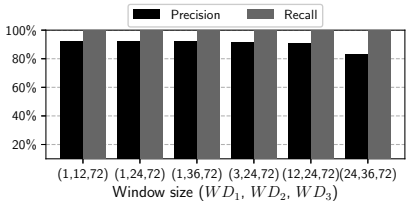

Figure 20: Effectiveness of Window Size.

region. Gandalf is effective in detecting these region-specific issues. For example, during one DiskRP service deployment, Gandalf made a no-go decision in SouthFrance, a late-stage production region. The alert turned out to be caused by a compatibility bug introduced by another component, which is only exposed after the latest DiskRP being deployed in this specific region. With the timely alerting and mitigation, only 3 subscriptions were impacted.

***Case III*: Latent Impact.** Gandalf detects not only immediate issues that happened right after the deployment but also latent issues that happen several hours or even days after deployments. Gandalf detects the latent issues in the early stage and prevents it from affecting customers in production environment. Figure 17 shows a deployment case of a Network Agent in Canary. OS crashed during 24 hours to 72 hours after the deployment and Gandalf issued a Sev2 alert (i.e., large customers impacts). The root cause was a conflict between old firmware version and new driver version. When the network agent rollout upgraded the NIC firmware and drivers, the firmware upgrade script missed one of the hardware revisions. Gandalf accurately attributed the faults to the Network Agent deployment even though the failures occurred 24 hours after the deployment, while many other concurrent updates was ongoing in these clusters.

5.4 Effectiveness of Correlation Algorithms

In this Section, we evaluate the effectiveness of the Gandalf correlation algorithms. The results are from real rollouts in Azure between Jan. 2018 to Nov. 2018.

Parameter Settings. w_i in Equation 3 is respectively set to 8, 4, 2, 1, so that the weights are exponentially decreased along the time windows. The spatial correlation threshold β is set to be 0.8 to tolerate noise in the telemetry data. The w_s in Equation 6 is set to 90 as the longest monitoring period of a rollout is 90 days. b is set to 1 and a is set to 0.2 so that the decay factor is scaled between $[0.2, 1]$.

Exponential Weights. Figure 18 shows that without the exponential weights (EW) used in temporal correlation for different time windows, the precision decreases but the recall remains the same. The reason is that given a spike of faults, without EW, the blame score is high for a bunch of components in addition to the real one, which leads to low precision but the same recall. EW treats component with different fault ages differently, which increase the precision.

Spatial Correlation. Figure 18 shows that by incorporating spatial correlation, Gandalf correlation precision is increased by 77%. The reason is that multiple components are often rolled out in similar time frame in a large system. The temporal correlation results can be very noisy. By incorporating the spatial correlation, the noisy results can be greatly reduced. As the algorithm can accurately identify the problematic components and prevent it from causing high impacts to customers, the recall also increases.

Time Decaying. Figure 18 shows that without the time decaying algorithm, the precision decreases by 58.8% and the recall decreases by 12.7%. The reason is that after a fault (*e.g.*, connection timeout) is detected in a bad rollout, the bug causing the fault will be fixed. Later, the same timeout fault might still occur due to other bugs in another component being deployed. If we do not have the time decaying algorithm, the fault may be still blamed on the old component while missing the faulty new component.

Veto. Gandalf uses a veto mechanism to reduce ambient noise. Figure 18 shows that without the vetos, the precision decreases by 8.7% and the recall rate is the same. The reason is that if the failure signature appears before the rollout, the failure is more likely not related to the rollout. If we only look at the failure after the deployment, we are more likely to stop rollout based on the ambient signals.

Accumulative Effects of Algorithms. Figure 19 shows the accumulative effect of different algorithms. Comparing Figure 19 with Figure 18, we can find that the spatial algorithm and the time decaying algorithm contribute most to precision. Although the individual algorithm such as veto, exponential weights is important as shown in Figure 18, without the spatial correlation algorithm and the time decaying algorithm, their effects alone on precision are relatively small.

5.4.1 Impact of Window Size

Figure 20 shows the effect of window size settings. We can see that the choices of window size do not significantly affect precision. This is because the main purpose of different windows is to differentiate the importance of the time between

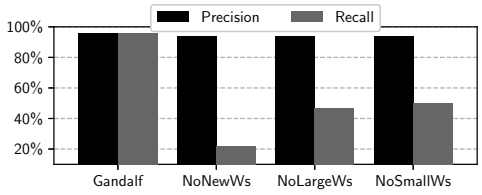

Figure 21: Impact of weight settings for control plane.

updates and anomalies. As long as the window size can provide different weights to the time interval between faults and update events, the precision should be similar as the spatial and temporal correlation algorithms are the major contributor to noise reduction. However, from the figure we can see that if WD_1 and WD_2 windows are too large, the precision decreases. The window size has no effects on the recall due to the same reason as that of the EW effectiveness.

5.4.2 Effectiveness of Weight Adjustments

Figure 21 shows how weight adjustments impact Gandalf decisions on the control-plane rollouts from 01/01/2019 to 03/06/2019. In this experiment, we compare Gandalf decisions using customized weights with 1) decisions without weight adjustment for all new faults (*NoNewWs*); 2) decisions without weight adjustments for all important faults (*NoLargeWs*), *i.e.*, all large weights changed back to 1; 3) decisions without weight adjustment for all noisy faults (*NoSmallWs*), *i.e.*, all small weights changed back to 1. We can see that *NoNewWs* decreases the precision slightly (1.8%) and decreases the recall significantly (73.2%). *NoLargeWs* and *NoSmallWs* have a similar effect on precision and recall. The experiments show that customizing weights can significantly improve recall while maintaining high precision.

From Sep. 2018 to Mar. 2019, 47% of the fault signatures are assigned with non-default weights, with 5 to 10 weights customized for a typical component/team. During this period, only the weights of 18 signatures in total are adjusted 4 times by developers. Thus, the tuning efforts overall are small.

6 Discussion

Gandalf has been running in Azure production environment for more than 18 months. In this section, we share perspectives from our users (Azure engineers and release managers) and lessons we learned along the way.

6.1 Transforming Deployment Experience

"We can call it a very good day for Gandalf!"

"This is a good case for Gandalf and a lesson or two for us"

"Gandalf has helped our rollout to the better. Thanks!"

– Comments from our users

The impact of Gandalf goes beyond accurately preventing bad rollouts. We are thrilled to witness how Gandalf has transformed the engineers and release managers' experience in deploying software changes:

From looking for scattered evidence to using a single source of truth. Before Gandalf was created, component-level watchdogs are used for safe deployment. These watchdogs only have isolated views about individual components. It is therefore difficult to rely on them for safe deployment. Consequently, release managers still need manual efforts to check the deployment behavior from additional data sources and communicate across related teams for ensuring deployment safety. The additional communication cost and decision overhead caused the deployment period to be long. Gandalf ingests comprehensive data sources in the data plane and control plane, and runs system-level analysis with anomaly detection and correlation models. Therefore, Gandalf can provide a single source of truth with various dimensions.

From skeptic to advocate. When some teams adopt Gandalf, the experienced engineers may be initially skeptical about Gandalf's data-driven approach and its decisions. As Gandalf detects complex failures that even experts can miss, the engineers start to trust Gandalf decisions and enforce the team to carefully investigate each "no-go" alert by Gandalf. For many teams, the deployment policy has become that the rollout will not continue to the next region unless Gandalf gives a green light decision.

From ad-hoc diagnosis to interactive troubleshooting. Before Gandalf, when an alert was sent to a component team, the engineers needed to write various queries for multiple data sources or access some sample nodes to grep the fault traces for diagnosis. Gandalf provides an interactive diagnosis portal to directly show the fault details. In particular, Gandalf aggregates the VM-level, node-level and cluster-level faults and buckets these faults based on their fault types. For example, faults "`Failed function: RuntimeVmBaseContainer::ValideXBlockBaseDisk: 0x81700035, XDiskLeaseIdMismatchWithBlobOperation: 0xc1425034`" will be bucketed into `ContainerFault-Creation-DiskLeaseIdMismatch`. Developers can drill down each fault bucket interactively to inspect the detailed fault source such as error messages and logs. The portal also shows the historical baselines to illustrate the difference so that the developers can better understand the impact scope and severity. Moreover, the portal shows the pivot analysis results, *e.g.*, SKU Gen2.3, that highlight potential causes.

6.2 Lessons Learned

We also learned several lessons while we built Gandalf. First, while F-score is a widely used metric to balance the precision

and recall of a decision model, in reality, different components may favor precision and recall differently. For teams with limited engineering capacity, they often prefer a system that only sends true alerts so that engineers can focus on investigating true issues. For teams that manage mission-critical services, 100% recall is a strict requirement. Missing any true issues causes much more damage than false alarms. A monitoring system should be tailored for different needs. For example, to fix the Meltdown and Spectre CPU vulnerability, the updates needed to be deployed to millions of nodes quickly. Since the rollout would impact millions of customers, Gandalf was optimized for extremely high recall and feature teams used the interactive portal to proactively monitor the rollout. False alarms were less critical in this scenario as engineer resources were enough to investigate all Gandalf reported issues.

Second, transparency and supporting evidence are crucial to build trust. It is difficult to trust machine decisions, especially on critical tasks. In cloud deployments, the release manager holds the same opinion because a simple false decision could be extremely harmful. That is why a black-box service that is not explainable is hard to be adopted for deployment monitoring even if the decisions are highly accurate. To gain trust, we design the Gandalf model to match the human decision process and make every step transparent. Gandalf surfaces rich supporting evidence, including the ranked list of faults, where the faults occurred, comparison of the time-series signal data before and after deployment, statistical summaries of the impact scope (*e.g.*, how many nodes and customers are affected). Such evidence helps explain to release managers why Gandalf makes each decision.

Third, analytics models should be adaptive. Many standard anomaly detection and time series algorithms are ineffective in a large-scale production system if applied without domain knowledge. It is almost impossible or at least extremely costly to learn such domain knowledge purely from the data. This is especially true when the system is constantly evolving, *e.g.*, an increasing number of new fault signals will emerge. We work closely with engineers to continuously incorporate their input into Gandalf decision model (because domain knowledge may not be fully discovered in one shot!).

7 Related Work

Time-series based anomaly detection models [13] provide high-quality alerts in DevOps. Instead of checking raw logs, DevOps can focus on the anomalous failure events while new build is rolling out [39]. Hangal and Lam first introduce DIDUCE [21], a practical tool that detects complex program errors and identifies root causes. Wang *et al.* [40] propose entropy-based anomaly testing, which uses arbitrary metrics distributions instead of fixed thresholds, for online systems anomaly detection. Fu *et al.* [17] propose classification algorithms to identify performance issue beacons. Laptev *et al.* [27] design a generic time-series anomaly detection framework, EGADS for Yahoo. Cohen*et al.* [16] use Tree-Augmented Naive Bayes models (TAN) to correlate SLO with system states as signatures. Panorama [23] detects gray failures through instrumenting observability hooks in the source code of observer components. However, without correlating anomalies with operational events, these work cannot identify which rollout is responsible or whether the detected anomalies are unrelated to deployments.

A number of tools have been built to analyze the correlation between KPI signals and system state changes. Bahl *et al.* [11, 12, 15] propose an inference graph that captures the dependencies between all components of the IT infrastructure by combining together these individual views of dependency and tries to locate the root cause. Azure is growing so fast that it is hard to build such dependency graph accurately at low cost. Several other methods are proposed for correlating systems signals and events [25, 32, 33, 41, 42]. But they mainly focus on extracting correlations in temporal dimension. Pure temporal correlation is insufficient for accurately identifying bad rollouts in our scenario.

8 Conclusion

In cloud infrastructures that undergo frequent changes, ensuring bad rollouts are accurately caught at the early stage is crucial to prevent catastrophic service outage and customer impact. In this paper, we present Gandalf, an end-to-end analytics service for safe deployment of cloud infrastructure. Gandalf assesses system-level impact of deployments by designing anomaly detection, correlation analysis and failure impact analysis algorithms in its decision model. It uses a lambda architecture to provide both real-time and batch deployment monitoring, with automated deployment decisions, a notification service and a diagnosis front-end. Gandalf has been running in Azure production for more than 18 months. Gandalf blocked 99.2% of the bad rollouts before they enter production. For data-plane rollouts, Gandalf achieved 92.4% precision with 100% recall. For control-plane rollouts, Gandalf achieved 94.9% precision and 99.8% recall.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd, Sujata Banerjee, for their valuable comments. We thank Microsoft Azure engineers who have been closely working with us, giving us feedback, adopting and using the Gandalf system, especially Chango Valtchev, Pengfei Huang, Alp Onalan, Francis David, Naga Govindaraju, Richard Russo, Roy Wang, Huaming Huang, Abhishek Kumar, Yue Zhao, Binit Mishra, Rishabh Tewari, Raza Naqvi, Igal Figlin, Anupama Vedapuri, and Cristina del Amo Casado. We thank Girish Bablani, Mark Russinovich, Dongmei Zhang, and Marcus Fontoura for their great support and sponsorship.

References

[1] Azure resource manager overview. https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview.

[2] Facebook 2.5-hour global outage. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919, 2010.

[3] Update on Azure storage service interruption. https://azure.microsoft.com/en-us/blog/update-on-azure-storage-service-interruption/, 2014.

[4] Google Compute Engine all-region downtime incident #16007. https://status.cloud.google.com/incident/compute/16007?post-mortem, 2016.

[5] Amazon Web Service suffers major outage, disrupts east coast internet. http://www.datacenterdynamics.com/content-tracks/colo-cloud/aws-suffers-a-five-hour-outage-in-the-us/94841.fullarticle, 2017.

[6] Lambda architecture. http://lambda-architecture.net/, 2017.

[7] Getting started with kusto. https://docs.microsoft.com/en-us/azure/kusto/concepts/, 2018.

[8] Service fabric. https://azure.microsoft.com/en-us/services/service-fabric/, 2018.

[9] H. Abdi. Discriminant correspondence analysis. In *Encyclopedia of Measurement and Statistic*, 2007.

[10] C. Aggarwal and C. Zhai. *Mining Text Data*, chapter A Survey of Text Clustering Algorithms, pages 77–128. Springer, 2012.

[11] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 13–24, Kyoto, Japan, 2007.

[12] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, pages 18–18, San Francisco, CA, USA, 2004.

[13] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[14] C. Chatfield. The Holt-Winters forecasting procedure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 27(3):264–279, 1978.

[15] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, E. Brewer, E. Brewer, and E. Brewer. Path-based faliure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 23–23, San Francisco, CA, USA, 2004.

[16] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 105–118, Brighton, United Kingdom, 2005.

[17] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie. Performance issue diagnosis for online service systems. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 273–278, Irvine, CA, USA, 2012.

[18] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, Miami, FL, USA, 2009.

[19] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, Santa Clara, CA, USA, 2016.

[20] 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.

[21] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, Orlando, FL, USA, 2002.

[22] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang. Identifying impactful service system problems via log analysis. 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, pages 60–70, Lake Buena Vista, FL, USA, 2018.

[23] 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. USENIX Association, October 2018.

[24] 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 '17, pages 150–155, Whistler, BC, Canada, May 2017.

[25] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 243–254, Barcelona, Spain, 2009.

[26] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[27] N. Laptev, S. Amizadeh, and I. Flint. Generic and scalable framework for automated time-series anomaly detection. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 1939–1947, Sydney, NSW, Australia, 2015.

[28] D. N. Lawley. A generalization of Fisher's z test. *Biometrika*, 30(1/2):180–187, 1938.

[29] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. STREAMSCOPE: Continuous reliable distributed processing of big data streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 439–453, Santa Clara, CA, USA, 2016.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[31] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems*, HotOS '19, Bertinoro, Italy, May 2019.

[32] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 613–622, Washington, DC, USA, 2010.

[33] C. Luo, J.-G. Lou, Q. Lin, Q. Fu, R. Ding, D. Zhang, and Z. Wang. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1583–1592, New York, New York, USA, 2014.

[34] K. Pearson. Notes on regression and inheritance in the case of two parents. In *Proceedings of the Royal Society of London*, 1895.

[35] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Seattle, WA, USA, 2014.

[36] N. Sahoo, J. Callan, R. Krishnan, G. Duncan, and R. Padman. Incremental hierarchical clustering of text documents. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 357–366, Arlington, VA, USA, 2006.

[37] T. Schlossnagle. Monitoring in a DevOps world. *Communications of the ACM*, 61(3):58–61, Feb. 2018.

[38] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, Shanghai, China, 2017.

[39] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *SIGSOFT Softw. Eng. Notes*, 40(2):1–4, Apr. 2015.

[40] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE Network Operations and Management Symposium*, NOMS '10, 2010.

[41] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, Z. Zang, X. Jing, and M. Feng. Funnel: Assessing software changes in web-based services. *IEEE Transactions on Services Computing*, Volume: 11:34–48, 2016.

[42] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 358–369, Hong Kong, China, 2002.