Aegis: Attribution of Control Plane Change Impact across Layers and Components for Cloud Systems

Xiaohan Yan [†], Ken Hsieh [†], Yasitha Liyanage [†], Minghua Ma [‡],
Murali Chintalapati [†], Qingwei Lin [‡], Yingnong Dang [†], and Dongmei Zhang [‡]
[†]Microsoft Azure [‡]Microsoft Research

***Abstract*—Modern cloud control plane infrastructure like Microsoft Azure has evolved into a complex one to serve customer needs for diverse types of services and adequate cloud-based resources. On such interconnected system, implementing changes at one component can have an impact on other components, even across different hierarchical computing layers. As a result of the complexity and interconnected nature of the cloud-based services, it poses a challenge to correctly attribute service quality degradation to a control plane change, to infer causality between the two and to mitigate any negative impact. In this paper, we present *Aegis*, an end-to-end analytical service for attributing control plane change impact across computing layers and service components in large-scale real-world cloud systems. Aegis processes and correlates service health signals and control plane changes across components to construct the most probable causal relationship. Aegis at its core leverages a domain knowledge-driven correlation algorithm to attribute platform signals to changes, and a counterfactual projection model to quantify control plane change impact to customers. Aegis can mitigate the impact of bad changes by alerting service team and recommending pausing the bad ones. Since Aegis' inception in Azure Control Plane 12 months ago, it has caught several bad changes across service components and layers, and promptly paused them to guard the quality of service. Aegis achieves precision and recall around 80% on real-world control plane deployments.**

***Index Terms*—Safe Deployment, Regression Detection, Impact Assessment, Counterfactual Analysis, Cloud Computing**

I. Introduction

As cloud computing continue to grow in scale and gain popularity [1]–[3], the cloud management system, also known as the control plane [4], is becoming more complex to provide services like resource creation, scheduling, and deletion. Each service is a component that is deployed in different computing layers throughout the cloud infrastructure. The control plane frequently gets deployed changes by the developers in order to address issues, add new features, and enhance performance [5]. Control plane changes may result in service interruptions, user annoyance, and significant financial loss due to the importation of defect code or configurations [6]–[9]. For example, AWS control plane outages, which impacted almost one-third of the public cloud market, had a negative impact on numerous companies and end users [10].

Conventionally, defects in changes are avoided by conducting various testing methods under a simulated environment [4], [11]. These approaches, however, often oversimplify the computing environment of a complex cloud system, leading to a gap between testing and production environments [12], [13]. To overcome the limitation of testing, component teams follow a safe change policy to conduct gradual rollout over batches of location units and enforce baking time between consecutive ones [5]. The hope is that any bad change would manifest in an early stage and get caught sooner to avoid a blast radius of impact. For example, a major hotfix of the control plane component can take place over 46 regions in five batches, each of which contains multiple regions, with 6-8 hours baking time between consecutive batches.

To guard change health, component teams often adopt local watchdogs and monitors to catch anomalous service performance [3], [14]–[24]. Some of the monitors are intelligent to attribute service anomalies to its own changes and provide guidance on investigation and mitigation. However, existing component-level monitors for many control plane components have their scope limited to the underlying service and computing layer. The lack of insights into changes and performance across components and layers introduces a blind spot in guarding change health for the entire control plane. However, it is non-trivial to attribute faults to changes across components and layers. For one, a naive approach that inspects the proximity between faults and changes would simply fail, because there is almost always some change right before a fault spike.

In Section II-C we discuss in detail the challenge posed by high concurrency level of changes. In addition, the choice of candidate fault signals and components for correlation requires domain knowledge to be distilled from the dependency structure shown in Figure 1. Finally, unlike data plane where changes occur at node level, control plane components deploy changes at a higher computing layer (e.g., region, zone, cluster). This results in magnitudes of difference for available change instances between data plane and control plane; many of the successful correlation practice on data plane (e.g., [5]) does not directly apply to a control plane scenario.

To address the aforementioned challenges, we introduce *Aegis*, an end-to-end analytical service for detecting and mitigating bad changes to ensure control plane service quality. Aegis consumes platform reliability and latency signals measured at different granularities (e.g., fleetwide, workload-specific, subscription-scoped signals), computing layers (e.g., region, zone) and service components (e.g., QoS[1] telemetry from CRP, NRP etc.). It correlates detected platform anomalies with

[1] Quality of Service (QoS) telemetry stores success and failure info of API calls for the underlying service component (e.g., ApiQosEvent for CRP)

software change events from relevant components and evaluates the strength of causality. Once a bad change is identified, Aegis takes actions to mitigate negative impacts by alerting the corresponding component team and, if applicable, automatically stop ongoing bad change via a change orchestration service.

Aegis at its core leverages a domain knowledge-driven correlation engine and a counterfactual-based impact evaluation engine. The correlation engine evaluates temporal and spatial fault prevalence pivoted upon a change and distills a correlation[2] score to indicate the strength of relevance between the change and platform anomalies. In correlation, Aegis accounts for domain knowledge about the computing system in several ways, including using customized weights to specify dependency knowledge between a fault type and a given component. To evaluate customer impact of a bad change, Aegis invokes its impact evaluation engine, which uses counterfactual projection methods to quantify the impact on both deployed and undeployed units. We will discuss the two core engines in greater detail in Section III-C and III-E.

As Aegis has been developed and running in Azure Control Plane over the past 12 months, it has caught several bad changes across control plane components, including CRP, NRP, RNM, AzSM and AzAllocator. The defects in these changes range from pure code bug that did not manifest during testing, to missing binaries during the change process, to broken contracts with another component. After catching these bad changes, Aegis automatically alerted corresponding teams with accurate attribution to change build version and prompted mitigated negative impacts by involving component team to revert their change or roll out a fix. Aegis contributes to Azure quality by complementing existing component-level monitors and providing a holistic view of fleet-wide performance.

To sum up, this work has the following major contributions:

- We describe how control plane component teams enable changes via code change or configuration change. We further motivate our work with technical challenges and formulate the main problem (Section II).
- We propose a novel approach named *Aegis* to attribute cloud control plane change impact via innovative techniques for fault correlation, decision validation and impact assessment (Section III).
- We demonstrate the business impact of Aegis through case studies of good catches and extensive experiments (Section IV and Section V).

II. Background and Motivation

A. Azure Control Plane

A dependable and high-performing cloud infrastructure plays a fundamental role in holding the quality bar for its service. A modern cloud computing system like Microsoft Azure employs a complex infrastructure to meet diverse needs on service and resource [5], [25], [26]. For example, a cloud computing infrastructure includes physical server devices organized in a hierarchical structure that consists of computing regions, zones, clusters, and nodes. Virtual services like virtual machines (VMs) or computing containers run under hierarchical layers to provide remote computing and storage functionality to customers. The physical complexity of the infrastructure is amplified by the interconnected nature of service components, each of which being loosely-coupled microservices for managing certain cloud resources. For example, Compute Resource Provider (CRP) is a component that processes requests from clients and other components (via APIs), such as requests for creating VMs or allocating resources. Figure 1 shows interactions among various service components under the current design of Azure compute architecture. Among the service components, those operating at region, zone or cluster levels constitute the *control plane*, which manages the infrastructure and provides interfaces for their functionalities. Below the control plane to node level, a *data plane* is made up of various plugins (called agents) running in a virtual machine hosting environment.

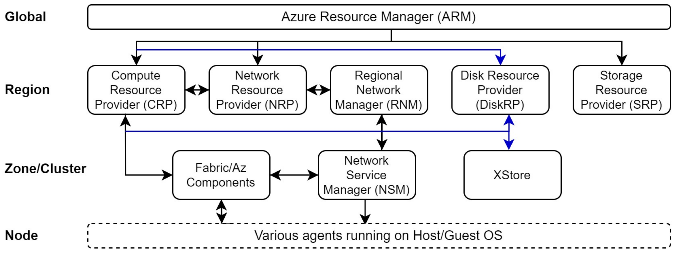
Global
Azure Resource Manager (ARM)
Region
Compute Resource Provider (CRP)
Network Resource Provider (NRP)
Regional Network Manager (RNM)
Disk Resource Provider (DiskRP)
Storage Resource Provider (SRP)
Zone/Cluster
Fabric/Az Components
Network Service Manager (NSM)
XStore
Node
Various agents running on Host/Guest OS

Fig. 1. Azure compute architecture with component dependency

B. Change Events in Azure Control Plane

An Azure control plane component has its underlying microservices loosely coupled to perform certain functionalities. For instance, CRP, Network Resource Provider (NRP) and Storage Resource Provider (SRP) manage their respective resources as their names suggest. By design, these control plane components operate at computing layers above nodes. When it comes to deploying changes, we categorize these components by computing layers of their deployments in Table I. These changes range from a regular code release to one-time-off bug fix to configuration change. The heterogeneity of these changes not only reflects in the diversity of deployed layer and nature of a change, but that variations exist even after scoping down to one component. For instance, CRP uses two branches to deploy non-test builds: the *release* branch and *hotfix* branch. CRP deployments to public regions are divided into two parts: the first rollout takes place from the release branch to six regions, followed by the second rollout taking place from the hotfix branch to the remaining regions. Additionally, CRP may deploy from the hotfix branch to selected regions to fix known issues. Note that we use *changes* and *deployments* interchangeably for the rest of the paper.

When comparing change events between control plane and data plane, the contrast of the amount of deployment instances

[2] Here *correlation* does not mean Pearson's correlation. We use *correlation* to indicate a generic process for evaluating the strength of relevancy between two entities (e.g., anomaly and change).

TABLE I
CONTROL PLANE COMPONENTS WITH CORRESPONDING DEPLOYMENT COMPUTING LAYERS.

Layer	Control Plane Component
Region	CPlat (CRP etc.), DiskRP, NRP, RNM etc.
Zone	AzSM, AzAllocator etc.
Cluster	NSM, XStore etc.

available for evaluation becomes prominent. Conceptually, the number of instances from a given deployment is determined by the amount of underlying deployment units. A data plane deployment typically occurs at the node level, meaning that the nodes gradually getting deployed are on the million scale. In contrast, control plane components deploy at higher layers and result in much fewer instances. To get a sense of resource partition above nodes, Azure currently has around 50 public regions, each of which is partitioned into three zones. In terms of clusters, there are a total of a couple of thousands of clusters from these public regions. This limitation makes it challenging to attribute platform degradation to a change event on control plane.

Because of the complexity and interconnected nature of the cloud system, component changes may introduce bugs or defects that impact not only its own service but also its dependent services. For example, in December 2021, an NRP's code change had an issue with name reservation for IP labels. The NRP release issue interrupted CRP's service, showing up as upticks in deleting VM failures due to networking errors. A bad change may also manifest across computing layers. For example, a zone-level service, AzAllocator introduced a bug in their rollout in East US Canary. The bug caused fabric failures when CRP tried to create VMs in the region. To guard the quality of control plane services, we propose to leverage all the health signals that are available across components and computing layers. To achieve it, we need to address some technical challenges that are described in the following section.

C. Challenges with System-wide Correlation

Following a safe deployment policy, component teams in Azure control plane conduct rollout gradually and employ component-level monitors to monitor platform health during and after a change. A typical component-level safe deployment monitor restricts its scope to the underlying service when it comes to picking fault signals (e.g., QoS telemetry) and change events (e.g., code and configuration changes) for coverage. For example, CRP's safe deployment monitor performs anomaly detection over CRP's ApiQosEvent and ProcessFailure signals and correlates detected anomalies with a nearby CRP rollout. The scope limitation hinders the capability of accurate attribution, because a platform anomaly, if were indeed caused by a bad change, is not necessarily caused by a CRP change. Naively extending the coverage of fault signals and change events across components and computing layers poses several risks, e.g., worsening correlation quality with the introduction of irrelevant noise. We identify the following challenges in conducting system-wide correlation that aims to break the boundaries of components and layers on control plane.

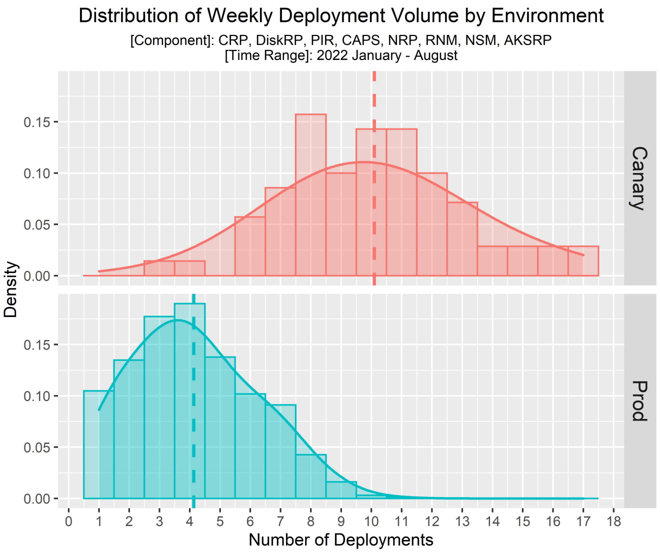

Fig. 2. Distributions of weekly deployment volume from eight components: (Top) canary environment and (Bottom) prod environment. As indicated by two dashed lines (for mean values), canary environment has higher deployment concurrency level than prod environment on average.

Challenge 1: Concurrent deployments across components increases the size of candidate pool and the difficult level of accurate attribution. Earlier environments like canary have even higher concurrency level than prod environment, making it challenging to catch an issue in canary.

In Figure 2, we compare the distribution of weekly deployment volume from eight control plane components between canary and prod environments. Using mean deployment volume to measure the deployment concurrency, we find canary's concurrency level is more than two folds of that of prod.

Challenge 2: Attributing platform anomalies to change events across computing layers requires innovation on methodology.

Existing correlation techniques used in component-level safe deployment monitors require both the fault signal and deployment signal to be measured over the same computing layer, e.g., both being measured at region level. These techniques do not directly apply to a cross-layer correlation scenario. An example of cross-layer correlation involves connecting a zone-level component change (e.g., AzAllocator's rollout) with region-level platform anomaly (e.g., CRP's upticks of fabric failures).

> **Challenge 3:** Domain knowledge should be systematically incorporated into the design of the attribution of control plane changes.

Domain knowledge plays an essential role in many successful analytical practices in Azure. The domain knowledge may be represented in various forms, but at the core it must reflect the cloud architecture (e.g., Figure 1). In practice, we abstract how Azure cloud computing system works into forms that can be readily incorporated into machine learning models as prior information. For example. Compute Artifact Publisher Service (CAPS), which is a CPlat service for publishers to publish images, is not involved in deleting VMs at all. This piece of domain knowledge should preclude our model from correlating deleting VM failures with a CAPS change.

> **Challenge 4:** The business need of knowing customer impact from a bad change requires intelligent impact evaluation techniques.

Finally, our cloud business needs to be able to answer questions like *”what is the customer impact from the bad change?”* and *”how much more impact would have been there had the bad change continue getting deployed?”* Component teams may have their own way of quantifying a regression impact to their service from the engineering perspective. However, gaps exist to connect the engineering side and our customers in understanding the regression impact on customer workloads and scenarios. We need to develop an impact evaluation model to answer these questions from the customer's standpoint.

III. Aegis System Design

To overcome the limitations of component-based monitoring strategy on Azure control plane, we create a novel regression attribution and mitigation service called *Aegis*, which addresses the technical challenges described in Section II-C. The capabilities of Aegis include: (1) connecting various platform health signals with changes across components and computing layers; (2) driving accurate attribution decision to changes with rich evidence; (3) mitigating negative customer impact by taking actions on bad changes. We use the following notations in the rest of the section to facilitate model discussion.

Notation: Let $\mathcal{F}(s) = \{F_t^u(s)\}_{u,t}$ denote a multi-dimensional time series of faults of type s over time t and location u. The fault type s can have different meanings for different parts of Aegis. Each $F_{t_j}^{u_i}(s) = \big(u_i, t_j, y_j^i(s)\big)$ encodes the location u_i, time t_j and fault metric value $y_j^i(s)$. We denote change events with a set of sequences $\mathcal{E} = \{E^{u_1}, E^{u_2}, ...\}$. Each $E^{u_i} = \big((t_1^i, c_1^i, v_1^i), (t_2^i, c_2^i, v_2^i), ...\big)$ is a sequence of pairs (t_j^i, c_j^i, v_j^i) where t_j^i is the time when the change event from component c_j^i of version v_j^i has occurred to the unit u_i, and $t_j^i < t_{j+1}^i$.

A. System Overview

As is shown in Figure 3, Aegis follows a top-down design and constitutes five subsystems: *Signal Collection*, *Correlation Engine*, *Validation Engine*, *Impact Evaluation Engine* and *Decision & Action Engine*. Each subsystem runs independently and continuously at its own cadence (e.g., every 15min, every hour) using a job scheduling service. Aegis consumes two kinds of signals from Azure control plane: platform fault telemetry and component changes events. The fault telemetry includes API call exceptions in the forms of failures and long latency. For change events, Aegis currently covers various code deployments and configuration changes on control plane. The input signals are processed by the Correlation Engine to evaluate relevancy for pairs of platform anomaly and change using a correlation model propelled by both ML algorithms and domain knowledge of the cloud system. By employing novel temporal and spatial correlation algorithms (see Section III-C1) and consolidating decisions at build level (see Section III-C3), Aegis successfully tackles Challenge 1 and attributes change impact with high confidence. The innovation in Section III-C4 addresses Challenge 2 by allowing Aegis connecting platform anomalies and changes among components residing on different computing layers.

Aegis adopts a Validation Engine to correct for known sources of environmental noise that may lead to incorrect correlation results (see Section III-D). For example, a planned powered-down drill may introduce noise in failures, leading to noisy correlations. Such noisy correlations are more easily removed as a downstream task as opposed to incorporating the drill signals into each input source. The Validation Engine also validates a correlation result using auxiliary info such as insights distilled from call stacks of correlated API calls. The use of the Validation Engine to validate correlation decisions with known noise in a cloud system serves as a solution to Challenge 3. As another example of incorporating domain knowledge, we embed fault-to-component relevancy as weights and factor the weights into correlations (see Section III-C2).

To address Challenge 4, an Impact Evaluation Engine consumes the correlation output to quantify customer impact of the correlated deployment, e.g., delta of operation failure rate before and after a deployment. The Impact Evaluation Engine at its core is a counterfactual projection model that models relationships among computing units (e.g., region-to-region relation) using historical data and projects a change to undeployed units.

The validated correlation result and customer impact result are processed by the Decision & Action Engine to drive a decision for each deployment and take actions if needed. For the deployments that receive a ”no-go” decision, the Action Engine notify corresponding component team with incident ticket. For ongoing deployments that Aegis decides to pause, the Action Engine emits its stopping signal to a deployment orchestration service to automatically pause the deployment. These mitigation actions avoid further customer pain and control the blast radius of a defect change.

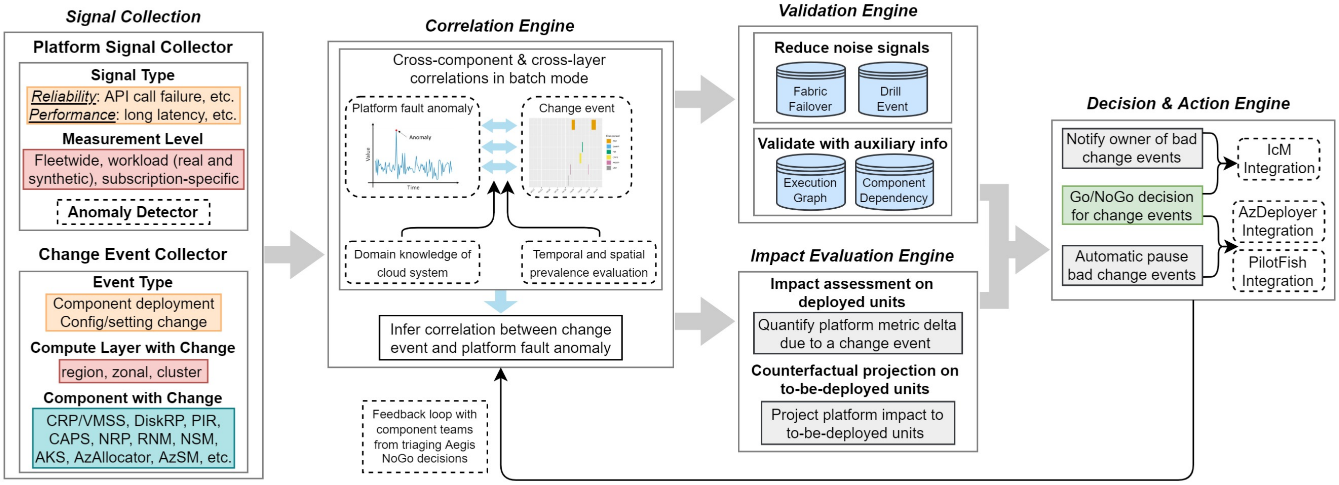
Signal Collection
Platform Signal Collector
Signal Type
Reliability: API call failure, etc.
Performance: long latency, etc.
Measurement Level
Fleetwide, workload (real and synthetic), subscription-specific
Anomaly Detector
Change Event Collector
Event Type
Component deployment
Config/setting change
Compute Layer with Change
region, zonal, cluster
Component with Change
CRP/VMSS, DiskRP, PIR, CAPS, NRP, RNM, NSM, AKS, AzAllocator, AzSM, etc.
Correlation Engine
Cross-component & cross-layer correlations in batch mode
Platform fault anomaly
Anomaly
Value
Time
Change event
Domain knowledge of cloud system
Temporal and spatial prevalence evaluation
Infer correlation between change event and platform fault anomaly
Feedback loop with component teams from triaging Aegis NoGo decisions
Validation Engine
Reduce noise signals
Fabric Failover
Drill Event
Validate with auxiliary info
Execution Graph
Component Dependency
Impact Evaluation Engine
Impact assessment on deployed units
Quantify platform metric delta due to a change event
Counterfactual projection on to-be-deployed units
Project platform impact to to-be-deployed units
Decision & Action Engine
Notify owner of bad change events
Go/NoGo decision for change events
Automatic pause bad change events
IcM Integration
AzDeployer Integration
PilotFish Integration

Fig. 3. Overview of Aegis System

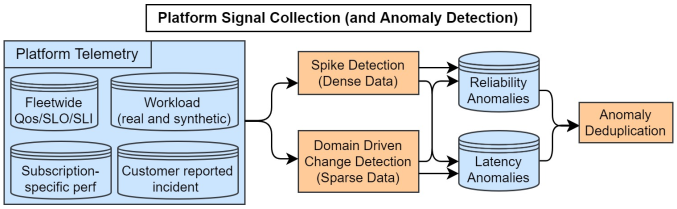
Platform Signal Collection (and Anomaly Detection)
Platform Telemetry
Fleetwide Qos/SLO/SLI
Workload (real and synthetic)
Subscription-specific perf
Customer reported incident
Spike Detection (Dense Data)
Domain Driven Change Detection (Sparse Data)
Reliability Anomalies
Latency Anomalies
Anomaly Deduplication

Fig. 4. Process Platform Health Signals into Anomalies

TABLE II
Platform Fault Anomaly Schema

Attribute	Description
TimeStamp	When the anomaly is detected
LocationId	Where the anomaly is detected
LocationLayer	Region, zone, cluster etc.
Signature	Fault signature of the anomaly
Value	Fault metric value of the anomaly
AdditionalInfo	Additional info of the anomaly

B. Signal Collections

Aegis consumes various platform fault signals from component service logs (e.g., QoS telemetry). These signals include API call failures and long latency instances, and are measured with different granularities (e.g., fleetwide, workload level and subscription level) and over different computing layers (e.g., region, zone, cluster). For the signals arriving as raw fault series, we perform anomaly detection on top as is shown in Figure 4. The choice of the anomaly detection model depends on several factors, e.g., the sparsity level of fault signals. A deduplication among all detected anomalies is needed to avoid double count. By design, the set of fault anomalies $\mathcal{A}$ can be treated as a subset of input fault signals $\mathcal{F}$, i.e., $\mathcal{A} \subset \mathcal{F}$. The output anomalies conform to the schema specified in Table II. Particularly, the $Signature$ of an anomaly encodes the anomaly type and other info that can categorize the underlying faults. For example, $Signature$ is defined as a tuple of $\{$operationName, resultCode, exceptionType$\}$ for CRP QoS failures. The fault metric $Value$ of an anomaly quantifies the severity and strength of the anomaly. Commonly used metrics include fault count and impacted subscription count of failures and latency measured at the $50^{th}, 95^{th}, 99^{th}$ percentiles.

The other signals being processed are change events collected over control plane components. The changes include regular code deployment to hotfix and configuration change. For some components, their changes can be readily processed from their service logs; for others, we get their change records from deployment orchestration service logs. The change event signals conform to the schema specified in Table III. Particularly, the $LocationLayer$ specifies the computing layer where a change gets deployed. Aegis currently considers deployments occurring at region, zone, and cluster levels.

TABLE III
Component Change Event Schema

Attribute	Description
TimeStamp	When the change event gets deployed
LocationId	Where the change event gets deployed
LocationLayer	Region, zone, cluster etc.
Component	Component deploying the change event
AdditionalInfo	Additional info of the change event

C. Correlation Engine

We conduct correlation analysis between platform anomalies and change events to evaluate their strength of relevancy and infer causal relationship between the two. Historically, only a subset of platform anomalies was caused by bad changes, while the remaining ones were due to transient issues, hardware issues etc. Using the following correlation techniques, Aegis can separate platform anomalies that are likely caused by bad changes from the rest and attribute the anomalies to respective changes with high confidence. The correlation engine assigns each deployment a correlation score whose value indicates the likelihood of the deployment having caused one or more platform anomalies. The engine at its core is an unsupervised learning model that inspects temporal and spatial fault prevalence around a deployment and ingests domain knowledge into the evaluation. A causal relationship may be derived by consolidating correlation results per component build version. We also describe how cross-layer correlation is achieved by the engine.

1) Temporal and Spatial Correlations: For simplicity of notation, we partition the s-typed platform faults by location units $\{u_i\}$:

$$\begin{aligned} \mathcal{F}(s) &= \{F^{u_1}(s), F^{u_2}(s), ...\} \\ \text{where } F^{u_i}(s) &= \left((t_1, y_1^i(s)), (t_2, y_2^i(s)), ...\right). \end{aligned} \tag{1}$$

Each $F^{u_i}(s)$ is a sequence of pairs $(t_j, y_j^i(s))$ where t_j is the time when s-typed faults with metric value $y_j^i(s)$ occur on unit u_i. Consider a change event that gets deployed to unit u_i,

$$e = \left(t_{(e)}^i, c_{(e)}^i, v_{(e)}^i\right) \in E^{u_i} \tag{2}$$

where $t_{(e)}^i$, $c_{(e)}^i$ and $v_{(e)}^i$ are the deployment time, component and build version for the change, respectively.

By design, Aegis detects anomalies $A^{u_i}(s)$ on top of $F^{u_i}(s)$, i.e., $A^{u_i}(s) \subset F^{u_i}(s)$. For correlation with change e, we focus on anomalies occurring after the change, i.e.,

$$A_{(e)}^{u_i}(s) = \left\{(t_j, y_j^i(s)) \in A^{u_i}(s) \middle| t_j > t_{(e)}^i\right\}. \tag{3}$$

An s-typed anomaly from unit u_i can be characterized with its time and metric value,

$$a = \left(t_{(a)}, y_{(a)}^i(s)\right) \in A_{(e)}^{u_i}(s). \tag{4}$$

In order to standardize the anomalies post the deployment of e, we compute $\mu^i(e,s)$ and $\sigma^i(e,s)$ as mean and standard deviation of the pre-deployment fault metrics:

$$\left\{y_j^i(s) \middle| (t_j, y_j^i(s)) \in F^{u_i}(s) \text{ and } t_j < t_{(e)}^i\right\}. \tag{5}$$

We compute a temporal correlation score for change e from (2) and an s-type anomaly a (that occurs after e) from (4) as

$$TS^i(e,a,s) = \frac{y_{(a)}^i(s) - \mu^i(e,s)}{\sigma^i(e,s)} \cdot df\left(t_{(a)}, t_{(e)}^i\right) \tag{6}$$

where $df\left(t_{(a)}, t_{(e)}^i\right)$ is a time-decay factor whose value exponentially decreases as the gap $(t_{(a)} - t_{(e)}^i)$ expands. For example, we may choose $df(t_1, t_2) = (1 + \exp(\alpha * (t_1 - t_2)))^{-1}$ where $\alpha > 0$ controls the decaying speed. The temporal correlation score encodes the temporal significance of s-typed anomalies after the change, compared to pre-deployment fault signals of the same kind. The score is higher if s-typed faults become more prominent after e gets deployed.

Temporal correlation by itself is prone to false positives when platform anomalies and a change happen to be close in time by coincidence. To curb the noise introduced by the temporal analysis, we inspect the same kind of faults across all units regardless of whether a unit has change e or not:

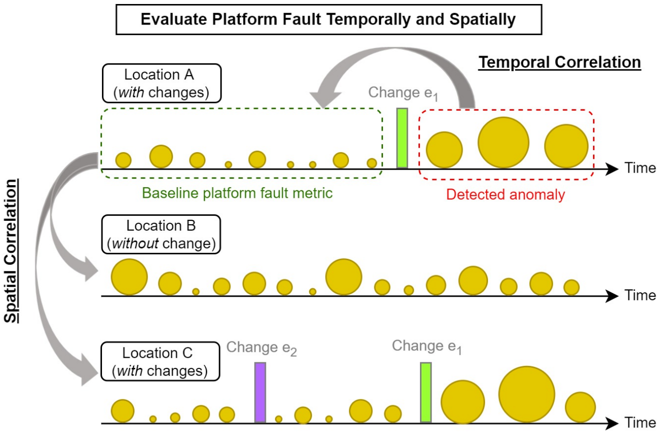

Fig. 5. Visualize temporal and spatial correlations. The circles correspond to faults with circle size representing the underlying fault metric value. The colored bars represent different changes.

$$M(e,s) = \left\{\left(t_j, y_j^{i'}\right) \in \bigcup_{i'} F^{u_{i'}(s)} \middle| |t_j - t_{(e)}^i| < \frac{w}{2}\right\} \tag{7}$$

where w is the window size for collecting the faults, e.g., $w = 14$day. Our practical observation is that anomaly a would no longer be prominent after comparing its metric value across all units for the same type of faults, if a was not caused by change e on unit u_i. We compute a spatial correlation score for change e and anomaly a as

$$SS^i(e,a,s) = \frac{Percentile\left(y_{(a)}^i(s), M(e,s)\right)}{100} \tag{8}$$

where $Percentile\left(y_{(a)}^i(s), M(e,s)\right)$ computes the percentile of anomaly metric value $y_{(a)}^i(s)$ among the entire pool of the same kind of faults $M(e,s)$. By definition, the spatial correlation score takes a value between 0 and 1. Anomalies that get correlated to innocent deployments out of pure temporal coincidence often receive a low spatial correlation score after inspecting the same kind of faults across all units. On the other hand, anomalies that indeed manifest a bad change are likely to retain its significance after spatial comparison by receiving a high spatial correlation score. The temporal and spatial correlations are illustrated in Figure 5.

2) Incorporate Fault Relevancy with Component into Correlation: We abstract the domain knowledge about the fault type, s, and the underlying component of the change, $c_{(e)}^i$, using a weight table of the following form:

$$W(s,c) = \begin{cases} \text{weight} > \text{default} & \text{if } c \text{ is relevant to } s \\ 0 < \text{weight} < \text{default} & \text{if } c \text{ is irrelevant to } s \\ \text{default weight} & \text{if else.} \end{cases} \tag{9}$$

For example, for CRP anomalies whose Signature containing `NetworkingInternalOperationError`, we assign higher-than-default weight values to networking components like NRP, RNM etc. Meanwhile, we assign minimal weight to irrelevant components to penalize the contribution of these anomalies to changes from those components.

To incorporate the domain knowledge in correlation, we combine the temporal correlation score in (6), the spatial correlation score in (8) and the weight table in (9) to get a single measurement of the strength of relevancy between the change and anomaly,

$$TSS^i(e,a,s) = TS^i(e,a,s) \cdot SS^i(e,a,s) \cdot W(s, c^i_{(e)}). \quad (10)$$

Finally, we derive an overall score for change e on unit u_i by aggregating the scores in (10) over all anomaly instances and all fault types, i.e.,

$$Score^i(e) = \sum_{\substack{a\in A^{u_i}_{(e)}(s)\\ s\in\{\text{all fault types}\}}} TSS^i(e,a,s). \quad (11)$$

By comparing $Score^i(e)$ with a pre-defined threshold, the correlation engine decides whether to flag change e on unit u_i. The threshold can be derived by training on historical catches, whose value shall reflect the noise levels of the unit and the fault signal.

3) Drive Build-Level Decision across Deployment Units: The above use of temporal and spatial correlations and domain knowledge would be sufficient for deriving accurate correlation result, when only one component participates in constructing the event set $\mathcal{E}$. However, system-wide correlation requires a holistic view of many components at a time, leading to concurrent deployments (i.e., deployments that occur simultaneously). As is discussed in Section II-C, high concurrency level poses a challenge in conducting high-quality correlation from a FDR (false positive rate) control perspective. On the other hand, a bad change may not necessarily manifest on all the units it gets deployed to, resulting in potential false negatives. For example, some change defects are setting-dependent, meaning that they would manifest on a fault telemetry only when certain setting requirement is met for the underlying unit.

Our solution is to consolidate correlation results across its deployment units and derive a view at the component build level, i.e., the $(c^i_{(e)}, v^i_{(e)})$ level for change e. Consider four component builds from control plane: (Component A, Build A.1), (Component B, Build B.1), (Component C. Build C.1) and (Component D, Build D.1). Among the four candidate builds, Build A.1 and Build B.1 have defects while the other two builds are of good quality. In Figure 6 we plot the distribution scores $Score^i(e)$ for these four builds across their respective deployed units indexed by i. The two defective builds have long right tails in the distributions of correlation scores, while the the innocent ones have scores concentrated at zero. By comparing scores across builds we can easily identify the bad ones, i.e., those with long right tails. In practice, we may define the flagging status of a component build by evaluating the significance of those score instances on the right tail among all the instances.

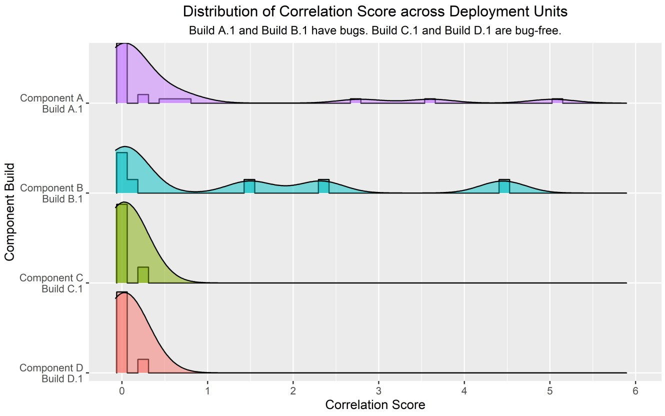

Fig. 6. Distribution of correlation score $Score^i(e)$ across deployed units $\{u_i\}$ for each component build of changes. Build A.1 from Component A and Build B.1 from Component B have defects; their score distributions are right tailed. The other two builds are innocent with scores concentrated at zero.

4) Correlation across Computing Layers: Another challenge with traditional correlation techniques happens when we try to correlate anomalies and changes across computing layers. Because of the complexity and interconnected nature of Azure system (see Figure 1), one component's defect change may interrupt its dependent component's service, where the two components may not operate at the same layer. For example, a CRP code bug may cause unavailability of certain VM provisioning service and manifest as spiked allocation failure rates on AzAllocator service. Reversely, an AzAllocator bug may manifest on CRP QoS as VM creation failures. In these examples, CRP and AzAllocator are regional and zonal services, respectively. In conducting system–wide correlation, we need to equip Aegis with the capability of connecting platform anomalies and changes across computing layers. We illustrate our two-fold solution in Figure 7. First, we identify key components in control plane and decompose fault signals $F^{u_i}(s)$ from a higher–layer component u_i into the granularity of a lower layer u_{ik}, such that $F^{u_i}(s) \triangleq \sum_k F^{u_{ik}}(s)$ where $u_i \triangleq \cup_k u_{ik}$. Then we compute the correlation score at the lower layer u_{ik} with (11). For example, we decompose CRP's regional fault signal into multiple zonal fault signals and correlate them with the AzAllocator's zonal deployments. The choice of components requires a solid understanding of control plane services. The signal decomposition is not trivial because it involves connecting multiple telemetries to get proper computing layer identifier for fault signals. Additionally, we may project changes $E^{u_{ik}}$ occurring at a lower computing layer u_{ik} to a higher layer u_i (such that $E^{u_i} \triangleq \cup_k E^{u_{ik}}$), so that the gap between computing layers is removed after the projection. Hence, we can compute the correlation score at the higher layer u_i with (11). For example, we may project cluster-level NSM deployments to the region level for correlating them with CRP's regional anomalies.

D. Validation Engine

The Validation Engine acts as a false positive filter between the Correlation Engine and the Action Engine. The engine

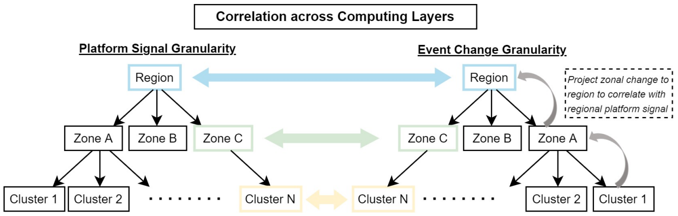
Correlation across Computing Layers
Platform Signal Granularity
Event Change Granularity
Region
Region
Project zonal change to region to correlate with regional platform signal
Zone A
Zone B
Zone C
Zone C
Zone B
Zone A
Cluster 1
Cluster 2
Cluster N
Cluster N
Cluster 2
Cluster 1

Fig. 7. For fault signals and change events that are available at the same layer, we directly conduct correlation between the two: region-to-region, zone-to-zone, cluster-to-cluster. For components residing at different layers, we may project changes from a lower layer to a higher one, e.g., zonal change to the region level, before conducting correlation at the higher layer. As an alternative, we may decompose fault signals of a high-layer component into ones at a lower layer (e.g., decompose CRP regional faults to zonal faults), before correlating with changes of a lower-layer component.

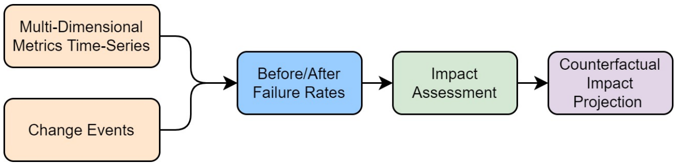
Multi-Dimensional Metrics Time-Series
Change Events
Before/After Failure Rates
Impact Assessment
Counterfactual Impact Projection

Fig. 8. Impact Evaluation Engine Architecture

consumes a set of build-level decisions and deems those that fail the validation as false positives. The post-correlation validation is accomplished via two independent checks. First, the engine checks if correlated anomalies are expected (e.g., due to by disaster recovery drill or fabric controller failovers). If a significant amount of the anomalies are explainable by known sources of noise, the correlated component build is shielded from being flagged to avoid false positives. The second check leverages auxiliary signals (e.g., call stack signals and services in Azure) to evaluate the accountability of the attributed component for its correlated fault anomalies. The engine mines insights from the call stack signals to determine if correlated anomalous failures originate from another component other than the attributed one. If that is the case, the attributed component build will be removed from taking further action to avoid false positives. After performing these validation checks, the Validation Engine pass its decisions on to the Action Engine, where only those attribution decisions that are not labelled as false positives are acted upon.

E. Impact Evaluation Engine

This section describes the framework for evaluating the impact of a bad change detected by the Correlation Engine. The proposed Impact Evaluation Engine quantifies the change impact in two steps: 1. impact assessment of the bad component version in deployed units, and 2. counterfactual projection of the impact on to-be-deployed units. In both steps, the engine models two input signals: a Multi-Dimension Metrics time series (which stores success and failure instances from underlying health telemetry) and change events (see Figure 8).

1) Impact Assessment of the Bad Change in Deployed Units: For s being an operation and u_i being a unit (e.g., region), we define the failure rate from time t_1 to t_2 in u_i as observed from pivot s as follows:

$$FR(u_i, s, t_1, t_2) = \frac{\sum_{t=t_1}^{t_2} \mathbb{1}_{\{y_t^i(s)=1\}}}{\sum_{t=t_1}^{t_2} \left(\mathbb{1}_{\{y_t^i(s)=1\}} + \mathbb{1}_{\{y_t^i(s)=0\}}\right)} \qquad (12)$$

where $\mathbb{1}_z$ is the indicator function of z (i.e., $\mathbb{1}_z = 1$ when z holds, and $\mathbb{1}_z = 0$ otherwise). Once the Correlation Engine detects a bad change (t^r, c^r, v^r) in region u_r, we derive the impact assessment as follows:

$$\begin{aligned} IA_1(u_r, s) &= FR(u_r, s, t^r, t^r + w_f) \\ &\quad - FR(u_r, s, t^r - w_b, t^r) \qquad (13) \\ IA_2(u_r, s) &= \frac{FR(u_r, s, t^r, t^r + w_f)}{FR(u_r, s, t^r - w_b, t^r)} \qquad (14) \end{aligned}$$

where $IA_1(u_r, s)$ and $IA_2(u_r, s)$ are absolute and relative the failure rate delta due to the bad component build version (c^r, v^r), respectively, and w_b and w_f are a look-back window and a look-forward window, respectively. We assess the impact in every unit where this bad component version (c^r, v^r) is deployed.

2) Counterfactual Projection of Impact onto To-Be-Deployed Units: Now that we quantify the impact of a bad change on deployed units, a nature question to ask is "*how much more impact would have been there had the bad change continue getting deployed?*" We answer this question with counterfactual analysis. Counterfactual analysis studies the impact of an intervention to the underlying entity by evaluating the counterfactual outcome (i.e., outcome had the intervention not happened). Counterfactual analysis involves identifying comparable entity without an exposure to the intervention, modeling exposed and non-exposed entities using pre-intervention data, and projecting counterfactual outcomes between exposed/non-exposed entities based on the learned model. The difference between the realized and projected outcomes allows one to quantify the impact of an intervention.

In our problem, the intervention is a deployed change on deployed units. We strive to model counterfactual platform performance (e.g., counterfactual fault rates) for to-be-deployed units, had the change not been paused and continued getting deployed to these units. To compute projected failure rates, we cannot naively transform failure rate bounds (e.g., failure rate percentiles) from deployed units linearly onto to–be deployed units, since doing so does not guarantee the projected failure rate ranging between 0 and 1. Instead, we model the failure rate using the sigmoid function in (15) for it restricts the failure rate to be within the range [0,1].

$$FR(u_i, s, t_1, t_2) = \frac{1}{1 + \exp^{-X(u_i, s, t_1, t_2)}}. \qquad (15)$$

The inverse sigmoid of the failure rate $X(u_i, s, t_1, t_2)$ can be equivalently expressed as

$$X(u_i, s, t_1, t_2) = \log \frac{FR(u_i, s, t_1, t_2)}{1 - FR(u_i, s, t_1, t_2)}. \qquad (16)$$

We compute the inverse sigmoid failure rate increment due to the bad change (t^r, c^r, v^r) in a deployed unit u_r as follows:

$$\Delta X(u_r, s) = X(u_r, s, t^r, t^r + w_f) - X(u_r, s, t^r - w_b, t^r). \tag{17}$$

To get a lower bound $\Delta_L X(u_r, s)$ and an upper bound $\Delta_U X(u_r, s)$ of the inverse sigmoid failure rate increment, we use the $p/2$th and $(100 - p/2)$th percentiles of $\{\Delta X(u_r, s)\}_r$, a set defined upon all deployed units $\{u_r\}$.

For each to-be-deployed unit u_k, we estimate a lower bound $\widetilde{FR}_L(u_k, s, t^r, t^r + w_f)$ and an upper bound $\widetilde{FR}_U(u_k, s, t^r, t^r + w_f)$ for the *would-have-been* failure rate had the change been deployed to u_k:

$$\begin{aligned}&\widetilde{FR}_L(u_k, s, t^r, t^r + w_f)\\ &= \frac{1}{1 + \exp^{-\left(X(u_k, s, t^r - w_b, t^r) + \Delta_L X(u_r, s)\right)}}\end{aligned} \tag{18}$$

$$\begin{aligned}&\widetilde{FR}_U(u_k, s, t^r, t^r + w_f)\\ &= \frac{1}{1 + \exp^{-\left(X(u_k, s, t^r - w_b, t^r) + \Delta_U X(u_r, s)\right)}}.\end{aligned} \tag{19}$$

Finally, we estimate the counterfactual impact of the bad change on u_k with lower bounds (i.e., $\widetilde{IA}_{1L}(u_k, s)$ and $\widetilde{IA}_{2L}(u_k, s)$) and upper bounds (i.e., $\widetilde{IA}_{1U}(u_k, s)$ and $\widetilde{IA}_{2U}(u_k, s)$) as follows:

$$\begin{aligned}\widetilde{IA}_{1L}(u_k, s) &= \widetilde{FR}_L(u_k, s, t^r, t^r + w_f)\\ &\quad - FR(u_k, s, t^r - w_b, t^r)\end{aligned} \tag{20}$$

$$\begin{aligned}\widetilde{IA}_{1U}(u_k, s) &= \widetilde{FR}_U(u_k, s, t^r, t^r + w_f)\\ &\quad - FR(u_k, s, t^r - w_b, t^r)\end{aligned} \tag{21}$$

$$\widetilde{IA}_{2L}(u_k, s) = \frac{\widetilde{FR}_L(u_k, s, t^r, t^r + w_f)}{FR(u_k, s, t^r - w_b, t^r)} \tag{22}$$

$$\widetilde{IA}_{2U}(u_k, s) = \frac{\widetilde{FR}_U(u_k, s, t^r, t^r + w_f)}{FR(u_k, s, t^r - w_b, t^r)}. \tag{23}$$

With these bounds, we end up with $(100 - p)$% confidence intervals, $(\widetilde{IA}_{1L}(u_k, s), \widetilde{IA}_{1U}(u_k, s))$ and $(\widetilde{IA}_{2L}(u_k, s), \widetilde{IA}_{2U}(u_k, s))$, for the failure rate increment in u_k had the change been deployed to u_k. In practice, both impact assessment and counterfactual projection are applied only to operations s whose failure rate increment is statistically significant from deployed units.

F. Decision and Action Engine

The final layer of Aegis is a Decision and Action Engine that orchestrates its ”go” or ”no-go” decision with rich evidence to component team. For each build-level no-go decision, Aegis invokes a notification service to automatically generates ticket incident to component team. For ongoing changes that are deemed with severe customer impact, Aegis emits a stopping signal to deployment orchestration services to automatically pause the change for component team to manually investigate. A fully automated service, the Decision & Action Engine guarantees prompt mitigation of detected deployment issues.

IV. Discussion

Aegis was developed and has been running on Azure control plane over the past 12months. In this section, we evaluate its business impact and provide sample catches in case studies.

A. Business Impact

Coverage and Scale Aegis currently covers 10 control plane services for their code rollouts, including CRP, NRP, and DiskRP, etc. It also covers dynamic configuration change from CRP. These services range from region-level services (e.g., CRP) to zone-level service (e.g., AzSM) to cluster-level service (e.g., NSM). Over an 8-month period from January 2022 to August 2022, Aegis has made decisions for 8000+ deployments from the underlying components.

We prioritized covering components that takes a central position in Azure stack (e.g., CRP), or on TDPR (Tenant Deployment Performance & Reliability) path, because of their great relevancy to key customer scenarios like single-instance VM create, concurrent VM create etc.

In terms of platform health signals, Aegis covers a diverse set of reliability and latency signals that are measured at various levels: fleetwide, workload level and subscription level. Major fleetwide signals under coverage include CRP and VM Scale Set (VMSS) faults from their QoS telemetry, which is processed at a scale of 1.2 million API call failures daily on average. In addition to fleetwide metrics, Aegis also covers real and synthetic customer workloads. For real workload, Aegis covers reliability and latency metrics from Fast Disk Attach & Detach workload. Synthetic workloads, like Aurora workloads, are meant to replicate customer representative usage of Azure service and are used to measure scenario experience and identify platform gaps. Aegis covers the complete set of 40+ Aurora synthetic workloads.

A Scale-Out Tool The design of Aegis allows the common infrastructure to be abstracted out, which greatly increases its scalability to cover more components and signals. Setting up a typical component-level deployment health monitor often requires the implementation of the entire end-to-end flow. In comparison, Aegis only requires adding signal-specific parts (e.g., correlation runner for a new signal) to complete the onboarding process. Aegis also provides a versatile interface that can support platform signals of various forms, as is evidenced by the diversity of the signals that Aegis covers.

Push Quality Left Component teams in Azure follow a safe deployment policy to deploy change. Before a change gets to production it needs to go through testing, stage, canary, pilot environments. When a bad change occurs, the north star is to catch it as early as possible. Aegis provides complementary coverage to component-level deployment health monitors, catching bugs that got missed by the first layer of protection. For example, in March 2022 Aegis caught a CRP bug that missed null-check handling when trying to remove a pipeline instance in CRP's release deployment. CRP's component-level monitor missed catching the bug because the bug only manifested under certain setting. With the Aegis' catch, CRP team stopped the rollout and retook a new release after fixing a bug. In practice, Aegis issued the majority of its no-go decisions in canary environment, which contributes to *Push Quality Left* in Azure.

B. Case Studies

Case 1: Cross-Component Impact In January 2022, RNM deployed build but missed a binary during the build process. Once the release build got deployed, it interrupted CRP's VMSS creation and manifested as upticks of VMSS creation failures with a specific exceptionType pointing to RNM issue. Aegis correctly correlated the failure upticks from CRP telemetry with the RNM rollout in East US Canary, and alerted RNM team. With Aegis' no-go signal, RNM team rolled back the build and fixed the bug. Using its impact evaluation techniques, Aegis found that the failure rate of VMSS create operation increased by 148 folds under the regressed RNM build, compared to before the rollout on the deployed region. For the remaining undeployed regions, Aegis projected failure rate of the same kind would increase by a range of (158, 169) folds at 95% confidence level, had the RNM build reached those regions. The prevented impact from the bad RNM deployment by Aegis demonstrates the value of the Aegis system.

Case 2: Cross-Layer Impact In April 2022, Aegis alerted AzSM team with their build rollout based on correlated CRP failures impacting VM creation and VMSS deletion in East US Canary. This decision was made using both cross-component correlation and cross-layer correlation techniques, for the AzSM rollout occurred at zone level. Aegis attributed CRP's fault spikes to AzSM, because the fault exceptionType suggested the failures were caused by fabric issue. Upon investigation by AzSM team, they found that AzCPGateway introduced a bug resulting in ArgumentOutOfRangeException when AzSM called its AzAllocator client, whose process involves AzCPGateway. To mitigate the impact, AzCPGateway downgraded to the previous version.This example demonstrates the challenge posed by the complexity of Azure architecture where many components depend on each other. In this catch, Aegis alerted AzSM because it did not cover AzCPGateway deployments. But the ticket successfully manifested the bug and led to actions after AzSM's investigation.

C. Generality of Aegis

Aegis at its core is a domain-knowledge driven attribution and impact evaluation system using platform observational data. Aegis can be generalized to other cloud or non-cloud scenarios as an end-to-end system to solve the attribution problem. Meanwhile, the success story of Aegis in Azure may lend its design philosophy in other applications, e.g., the adoption of Validation Engine to reduce noise in preliminary decisions from the Correlation Engine.

The utilization of domain knowledge proves to be a key to Aegis success by improving the explainability of a result in the context of Azure cloud system. The abstraction of domain knowledge may vary from application to application, but they share a commonality that the domain knowledge shall encode any prior knowledge about the system. While Aegis provides some concrete examples of how domain knowledge is reflected in the system (e.g., customized weights, validation with known noise), it certainly does not restrict the specific form and use of domain knowledge in a different application. The ultimate goal with the use of domain knowledge is to make the results not only correct from a machine learning perspective, but also relevant in the context of an application.

V. Experiment

We evaluate the performance of Aegis using real code rollouts from three Azure control plane components (CRP, RNM and AzSM) between January and March of 2022. During the time, the three components deployed over 60+ unique build versions (i.e., unique v's) via 700+ deployment instances (i.e., all changes e across units) at regional and zonal levels. Among the components, CRP and RNM are regional services from Compute and Network, respectively, while AzSM is a zonal service. For simplicity we use reliability signals from CRP fleetwide (i.e., ApiQosEvent) as the only source of fault for correlation. We quantify Aegis performance in real world using experiment results and show how the performance varies as we adjust some of the novel practices in Aegis (e.g., the adoption of Validation Engine, the use of fault signature weight). Because of the difference in the scope of coverage (e.g., Gandalf [5] on control plane is enabled as a component-level monitor hence it only covers deployments from the underlying component), we do not compare the performance between Aegis and Gandalf here.

We collect pre-deployment faults in (5) over a 21-day window before every rollout. The time-decay factor in (6) is set to $df(t_1, t_2) = ((t_1 - t_2)/6hour + 1)^{-1}$. The window size w in (8) is set to 14 days. The default value of fault signature weight $W(s, c)$ from (9) is set to 1, with higher-than-default and lower-than-default weights being set to 10 and 0.01, respectively. The threshold for the aggregated score from (11) is determined to reflect the noise level of ApiQosEvent over the underlying environment of unit u_i. In this experiment, the canary environment is set with a threshold that is roughly 4 times that of the production environment. Aegis conducts validation checks using Drill signal and Execution Graph in its Validation Engine.

The performance of Aegis is summarized in Figure 9. With all the features on, Aegis achieves roughly 80% for both precision and recall on the data set. The false negatives are missed because bugs in the bad rollouts manifested on another platform fault signal that is not covered by Aegis in this experiment. Had Aegis covered all the relevant fault signals, the recall would have been perfect. From Figure 9, removing the Validation Engine introduces false positives that are avoidable with Drill and Execution Graph. The precision decreases by 28.6% when removing the Validation Engine. The recall does not change with or without the Validation Engine, indicating that the engine does not introduce any false negative. When removing signature weight $W(s, c)$ from Aegis, both the precision and recall are greatly impacted: precision drops by 37.5% and recall almost halves. The worsening decision quality without signature weight shows the importance of reflecting

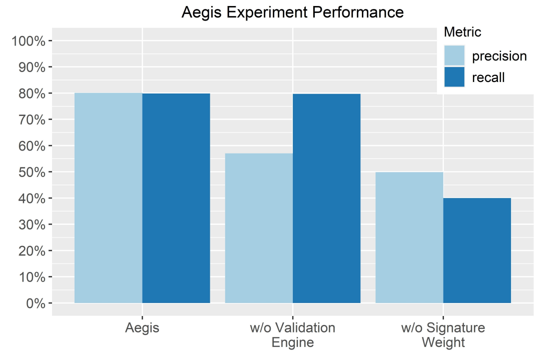

Fig. 9. Aegis performance on real rollouts from Azure control plane (CRP, RNM and AzSM) between January and March of 2022. Aegis made decisions on the rollouts using CRP fleetwide reliability signal. The delta of precision and/or recall shows the value of key practices in Aegis, including the adoption of Validation Engine and the use of Signature Weight in conducting correlation.

domain knowledge about component and fault relevance in conducting correlation analysis.

VI. Related Work

Impact assessment of change event. The impact of change event has attracted considerable attention in cloud computing systems. Previous work conducts time series anomaly detection approach, such as PCA [27], CUSUM [28], iSST [29], and LSTM [30], to detect change events based on the monitoring data, However, all these approaches focus on the change event detection rather than the change event attribution between the change event and its behaviors reflected in the monitoring data. Funnel [31], [32] adopts difference in differences, and Gandalf [5] use spatial and temporal correlation to blame bad changes. The correlation among service components and the correlation at multiple granularities, which are common and complex in cloud computing systems, are not considered by these correlation approaches.

Counterfactual analysis. Counterfactual analysis can establish a causal relationship between treatments and results [33], which has been used in many domains. For example, DBShelock [34] used a causal model to predict database performance issues. Diagnosing missing events in distributed systems adopts the provenance graph [35], [36]. These studies adopt the concept of counterfactual analysis in different scenarios, and we also illustrate the importance of using counterfactual analysis to attribute control plane changes in the cloud systems.

VII. Conclusion

Under the setting of large-scale cloud infrastructure, we present the Aegis system which is an end-to-end analytical service for attributing and mitigating change event impact across computing layers and service components on Azure control plane. Aegis follows a top-down design and leverages a domain knowledge-driven correlation algorithm to attribute platform signals to change events, and a counterfactual projection model to quantify change event impact to customer. Over the past 12 months, it has caught several bad deployments across control plane components and layers and guarded Azure service for its customers.

VIII. Acknowledgement

The authors thank Ryan Lingg, Tewbesta Alemayehu, Samiul Saeef, Theo Shiao, Prateek Gangwal, Daniele Maso, Ze Li and Randolph Yao for their contribution through their valuable feedbacks.

References

[1] Amazon, "Using cloudwatch anomaly detection," 2017, https://docs.aws.amazon.com/cloudwatch/index.html.

[2] 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*. ACM, 2019, pp. 136–141.

[3] X. Zhang, J. Kim, Q. Lin, K. Lim, S. O. Kanaujia, Y. Xu, K. Jamieson, A. Albarghouthi, S. Qin, M. J. Freedman *et al.*, "Cross-dataset time series anomaly detection for cloud systems," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 1063–1076.

[4] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 143–159.

[5] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy *et al.*, "Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 389–402.

[6] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "Continuous incident triage for large-scale online service systems," in *Proceedings of the 34th ASE International Conference on Automated Software Engineering*. IEEE/ACM, 2019, pp. 364–375.

[7] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "An empirical investigation of incident triage for online service systems," in *Proceedings of the 41st ICSE-SEIP International Conference on Software Engineering: Software Engineering in Practice*. IEEE/ACM, 2019, pp. 111–120.

[8] Z. Chen, Y. Kang, L. Li, X. Zhang, H. Zhang, H. Xu, Y. Zhou, L. Yang, J. Sun, Z. Xu *et al.*, "Towards intelligent incident management: why we need it and how we make it," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1487–1497.

[9] 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*, 2016, pp. 1–16.

[10] T. McCarthy, "Aws outage brings dr strategies back into focus," https://www.techtarget.com/searchdisasterrecovery/news/252511325/AWS-outage-brings-DR-strategies-back-into-focus.

[11] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020.

[12] C. Lou, C. Chen, P. Huang, Y. Dang, S. Qin, X. Yang, X. Li, Q. Lin, and M. Chintalapati, "Resin: A holistic service for dealing with memory leaks in production cloud infrastructure," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 109–125.

[13] 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)*, 2018, pp. 1–16.

[14] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at microsoft," in *Proceedings of the 25th SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2019, pp. 3009–3017.

[15] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 27th ESEC/FSE Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 807–817.

[16] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, "Uniparser: A unified log parser for heterogeneous log data," in *Proceedings of the Web Conference*. ACM, 2022, p. 1893–1901.

[17] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu *et al.*, "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017, pp. 1–10.

[18] C. Zhang, D. Song, Y. Chen, X. Feng, C. Lumezanu, W. Cheng, J. Ni, B. Zong, H. Chen, and N. V. Chawla, "A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1409–1416.

[19] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang *et al.*, "Prefix: Switch failure prediction in datacenter networks," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, pp. 1–29, 2018.

[20] H. Xu, W. Chen, N. Zhao, Z. Li, J. Bu, Z. Li, Y. Liu, Y. Zhao, D. Pei, Y. Feng *et al.*, "Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications," in *Proceedings of the 26th WWW World Wide Web Conference*, 2018, pp. 187–196.

[21] D. Park, Y. Hoshi, and C. C. Kemp, "A multimodal anomaly detector for robot-assisted feeding using an lstm-based variational autoencoder," *Robotics and Automation Letters*, vol. 3, no. 3, pp. 1544–1551, 2018.

[22] B. Zong, Q. Song, M. R. Min, W. Cheng, C. Lumezanu, D. Cho, and H. Chen, "Deep autoencoding gaussian mixture model for unsupervised anomaly detection," *International Conference on Learning Representations*, 2018.

[23] M. Ma, S. Zhang, J. Chen, J. Xu, H. Li, Y. Lin, X. Nie, B. Zhou, Y. Wang, and D. Pei, "Jump-starting: Multivariate time series anomaly detection for online service systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 413–426.

[24] M. Ma, Z. Yin, S. Zhang, S. Wang, C. Zheng, X. Jiang, H. Hu, C. Luo, Y. Li, N. Qiu *et al.*, "Diagnosing root causes of intermittent slow queries in cloud databases," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1176–1189, 2020.

[25] M. Ma, Y. Liu, Y. Tong, H. Li, P. Zhao, Y. Xu, H. Zhang, S. He, L. Wang, Y. Dang, S. Rajmohan, and Q. Lin, "An empirical investigation of missing data handling in cloud node failure prediction," in *Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, pp. 1453 – 1464.

[26] Y. Liu, H. Yang, P. Zhao, M. Ma, C. Wen, H. Zhang, C. Luo, Q. Lin, C. Yi, J. Wang *et al.*, "Multi-task hierarchical classification for disk failure prediction in online service systems," in *Proceedings of the SIGKDD Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2022, pp. 3438–3446.

[27] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," *ACM SIGCOMM computer communication review*, vol. 35, no. 4, pp. 217–228, 2005.

[28] A. A. Mahimkar, H. H. Song, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and J. Emmons, "Detecting the performance impact of upgrades in large operational networks," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, pp. 303–314.

[29] M. Ma, S. Zhang, D. Pei, X. Huang, and H. Dai, "Robust and rapid adaption for concept drift in software system anomaly detection," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 13–24.

[30] N. Zhao, J. Chen, Z. Yu, H. Wang, J. Li, B. Qiu, H. Xu, W. Zhang, K. Sui, and D. Pei, "Identifying bad software changes via multimodal anomaly detection for online service systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 527–539.

[31] S. Zhang, Y. Liu, D. Pei, Y. Chen, X. Qu, S. Tao, and Z. Zang, "Rapid and robust impact assessment of software changes in large internet-based services," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015, pp. 1–13.

[32] 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 Service Computing*, 2016.

[33] K. von Prince, "Counterfactuality and past," *Linguistics and Philosophy*, vol. 42, no. 6, pp. 577–615, 2019.

[34] D. Y. Yoon, N. Niu, and B. Mozafari, "Dbsherlock: A performance diagnostic tool for transactional databases," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1599–1614.

[35] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo, "The good, the bad, and the differences: Better network diagnostics with differential provenance," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 115–128.

[36] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo, "Diagnosing missing events in distributed systems with negative provenance," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 383–394, 2014.