Harmonia: A Continuous Service Monitoring Framework Using DevOps and Service Mesh in a Complementary Manner

Haan Johng[1(✉)], Anup K. Kalia[2], Jin Xiao[2], Maja Vuković[2], and Lawrence Chung[1]

[1] University of Texas at Dallas, Richardson, TX 75080, USA
{haanmo.johng,chung}@utdallas.edu

[2] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
{Anup.Kalia,jinoaix,maja}@us.ibm.com

Abstract. Software teams today are required to deliver new or updated services frequently, rapidly and independently. Adopting DevOps and Microservices support the rapid service delivery model but leads to pushing code or service infrastructure changes across inter-dependent teams that are not collectively assessed, verified, or notified. In this paper, we propose Harmonia - a continuous service monitoring framework utilizing DevOps and Service Mesh in a complementary manner to improve coordination and change management among independent teams. Harmonia can automatically detect changes in services, including changes that violate performance SLAs and user experience, notify the changes to affected teams, and help them resolve the changes quickly. We applied Harmonia to a standard application in describing Microservice management to assist with an initial understanding and strengths of Harmonia. During the demonstration, we deployed faulty and normal services alternatively and captured changes from Jenkins, Github, Istio, and Kubernetes logs to form an application-centric cohesive view of the change and its impact and notify the affected teams.

Keywords: DevOps · Service Mesh · Microservice · Monitoring · Enterprise Cloud Management

1 Introduction

Software teams today are required to deliver new or updated services frequently, rapidly and independently. They look to DevOps to increase speed and frequency of service delivery by automating testing and deployment of services. Microservices, an architectural concept consisted of small-sized services that are independently deployable, scalable, and manageable, further helps the software teams to deliver services in a more rapid, incremental, and independent manner.

© Springer Nature Switzerland AG 2019
S. Yangui et al. (Eds.): ICSOC 2019, LNCS 11895, pp. 151–168, 2019.
https://doi.org/10.1007/978-3-030-33702-5_12

For example, Amazon and Netflix deploy thousands of times per day by using DevOps and Microservices [6,15].

Although adopting DevOps and Microservice design brings the aforementioned benefits, it also brings communication and collaboration challenges among independent software teams collectively contributing to changes in services over time. As each team pushes code or service infrastructure changes into the environment, its impacts on inter-dependent teams are not collectively assessed, notified, or verified. Rather they are present ad hocly across multiple data sources: code changes and commits can be detected through git, deployment configuration changes are visible through DevOps pipeline, runtime performance issues can be reported by Istio or Kubernetes depending on what is monitored. However, there is no correlation across changes in an application code, its configuration and runtime performance, especially when multiple independent services and development teams are involved. As a result, integration errors, misconfigurations, and security exposures may occur that are difficult to detect and trace across teams and resolve in a timely manner. Today's approach to detect and diagnose issues caused by changes through performance diagnosis or root-cause analysis is therefore time-consuming and reactive.

In this paper, we propose *Harmonia*[1] - a continuous service monitoring framework utilizing DevOps and Service Mesh in a complementary manner, as among the first of its kind to the best of our knowledge, to improve coordination and change management among independent teams. Harmonia can automatically detect code and infrastructure changes in services, including changes in code, configuration, deployment, and application performance. Furthermore, Harmonia can assess the impact of the changes to other services, and notify the changes to affected teams, whereby helping software teams resolve the changes quickly. More specifically, Harmonia offers an ontology alignment between DevOps logs and Service Mesh logs to utilize service deployment information from DevOps together with service run-time interaction information from Service Mesh. Harmonia supports declarative rules for detection and notifications to define what to detect and notify and whom to be notified. Thus, Harmonia takes a proactive approach to change management whereby defects and performance issues are detected as they occur, their impact across service components are assessed, and actions are taken by notifying both the teams (or team members) accountable for the change as well as impacted by the change.

To demonstrate Harmonia, we have built a capability to correlate logs from Github and Jenkins for DevOps and logs from Istio and Kubernetes for Service Mesh. We applied Harmonia to the Bookinfo application, which is a standard application in describing Istio for Microservices management and ran a simulation to comprehend the applicability of Harmonia and its strengths. During the simulation, we deployed faulty and normal services at regular intervals to observe whether Harmonia is able to capture the changes and notify them to the impacted teams correctly automatically. Our demonstration shows that, compared to existing DevOps and Service Mesh frameworks, Harmonia offers a

[1] Harmonia is the goddess of harmony and concord in Greek mythology.

better interpretability for software teams regarding service changes, the impact of the changes, and source of the changes in a timely manner, by representing service changes both in service deployment phase and in the run-time phase in a single view.

The rest of the paper is organized as follows, Sect. 2 provides related work, and Sect. 3 describes Harmonia. Section 4 presents our demonstration, followed by observations and discussions. In the end, a summary of contributions and future work are described.

2 Related Work

We discuss the related work on monitoring and root-cause analysis on microservices and DevOps.

In terms of monitoring, Heinrich et al. [9] highlight research directions in microservices with respect to performance-aware testing, monitoring and modeling services. Specifically they emphasize that due to frequent releases, extensive system and integration tests are not possible. Although canary deployments take care some aspect of the problem by releasing the deployment to a few set of users, however, such deployment process can be expensive and time-consuming. Pina et al. [19] propose an approach to monitor microservices by decoupling monitoring functionalities from function-oriented microservices. For monitoring they use Zuul an adapted gateway from Netflix. Fadda et al. [4] provide an approach to support microservices deployment in multi-cloud environments emphasizing on the quality of monitoring. Their proposed approach creates a knowledge base that mediates between the perspectives of the cloud provider and the application owner and a Bayesian network that enhances the provider's monitoring capabilities. Haselböck and Weinreich [8] propose guidance models for monitoring microservices. The models are derived from literature, previous work on monitoring distributed systems and microservice-based systems. Phipathananunth and Bunyakiati [18] provide Pink a framework that monitors microservices to assess non-functional properties such as session management, caching and security. The major focus with such monitoring base contributions is primarily tied to monitoring a service mesh. Such contributions do not connect service mesh with DevOps that have additional information on deployment. In case of service anomalies or abnormalities, such approaches may not trace microservices that might get impact nor can notify the teams in charge of the services together with recent deployment and program code change history on the services to help the teams to plan mitigation actions instantly.

In terms of root cause analysis, Lin et al. [16] propose Microscope to detect abnormal services with a ranked list of possible root causes. Wang et al. [20] propose CloudRanger that constructs causal graphs to determine the culprit services that are responsible for cloud incidents. Myunghwan et al. [14] provide MonitorRank that monitors historical and current time-series metrics of each sensor as its input along with the call graphs generated between the sensors to create an unsupervised model for ranking. Chen et al. [3] propose CauseInfer

that creates a two-layer hierarchical causality graph from a distributed system to infer the root causes along the causal paths. Jayathilaka et al. [10] propose Roots that monitors a full-stack application to determine the root cause for an anomaly. It does so by analyzing previous workload data of the application and the performance of the internal PaaS services on which the application depends. Existing approaches to determine root causes are reactive based approaches, i.e., they identify a root cause after an anomaly has occurred. Also the current approaches do not consider logs generated from DevOps to pin point who is accountable for the root cause.

In DevOps, most of the contributions emphasize on utilizing microservices that facilitate rapid deployments of services. For example, Balalaie et al. [1] emphasize on how monolithic architecture can be broken down in to microservices considering microservices can quickly adapt to technological changes, reduce time-to-market and provision a better development team structuring around services. Zhu et al. [22] describe how DevOps can reduce time between committing a change to a system and the change being productionized ensuring high quality. Brunnert et al. [2] provide performance-relevant aspects of DevOps concept. Fitzgerald and Stol [5] propose BizDev that continuously assess business strategy and software development. Gupta et al. [7] propose an approach to automatically discover execution behavior models for the deployed and the new version using the execution logs. However, there seems a lack of studies on DevOps using service mesh that provides monitoring information of microservices and interactions among them. Without monitoring information, it is challenging to estimate the potential degree of impacts on other services before deploying updates and analyze the actual impacts after the deployment.

3 Harmonia - A Continuous Service Monitoring Framework

Harmonia is a continuous service monitoring framework that aims to reduce delays in communications among independent software teams regarding changes in services and impacts of the changes. Figure 1 shows the ontology of Harmonia. By using DevOps and Service Mesh in a complementary manner, Harmonia monitors changes in services by detecting violations of service level agreements (SLAs, e.g., latency SLA) together with impacted services after the changes are pushed. Harmonia further notifies software teams to assist them to take appropriate actions to remediate their services from the impact. The Harmonia rules define the following: *what to detect and notify* and *whom to be notified*. Each rule acts as a reference to let software teams customize the rule for their applications.

The underlying process in Harmonia consists of three steps as described in Fig. 2. In the first step, Harmonia aligns the ontologies obtained from the service deployment information i.e., from DevOps logs with run-time service interaction information obtained from Service Mesh logs to create an integrated body of knowledge. In the second step, once Harmonia detects changes in services from

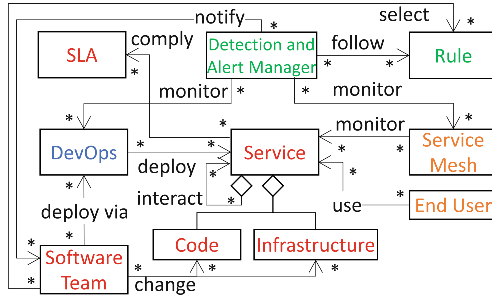


Fig. 1. The ontology of Harmonia for detecting and notifying changes in services.

DevOps logs, it traces the dependent services that might have recently interacted with the changed services. Harmonia does so by monitoring Service Mesh logs. Based on the detected changes, Harmonia notifies the changes to associated software teams that own the dependent services. In the third step, Harmonia detects SLA violations on services by monitoring Service Mesh logs after changes are pushed. Then, Harmonia traces recent changes on the problematic services, which are potential causes of the SLA violations, and notifies the change information to associated software teams.

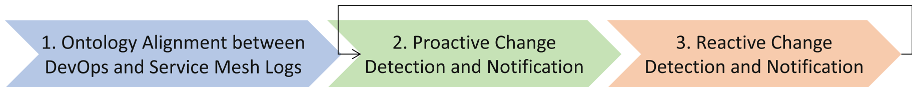


Fig. 2. The underlying process in Harmonia for detecting and notifying changes in services.

3.1 Ontology Alignment Among DevOps and Service Mesh Logs

DevOps is a framework to automate deployment and testing of services from development environments to production environments. Service Mesh is a framework for monitoring and managing interactions among (micro-) services. DevOps logs contain service deployment information such as logs for code push and service deployment. However, such logs do not include run-time service interaction information such as communications among services, latency between services and so on. On the other hand, Service Mesh logs contain run-time service interaction information but do not contain service deployment information. Without having the deployment information and run-time interaction information in a single view, software teams as of now manually inspect the impacts of changes, e.g., latency SLA violation, the source of such changes, software teams that might be impacted by the changes, their contacts, and notify them accordingly. Overall such process is time-consuming and the resultant delay in communication to appropriate software teams might delay the possible mitigation, thereby hurting the goal of frequent service delivery to production environments.

To create an integrated body of knowledge from DevOps and Service Mesh logs, we extract ontologies from both the logs and align them by common attributes. Work on log mining has researched in various domains [17]. For utilizing logs, it is essential to extract ontology, which is a set of important concepts, relationships among the concepts, and constraints, to understand what knowledge to utilize from the logs [11–13,21].

We extract the DevOps ontologies from Jenkins logs and the Service Mesh ontologies from Istio logs as shown in Fig. 3. Note that different DevOps and Service Meshes can produce different ontologies. Thus, the ontology is *a* reference ontology and may not generalize to other frameworks. Nonetheless the underlying methodology to extract and align the onotologies remains the same.

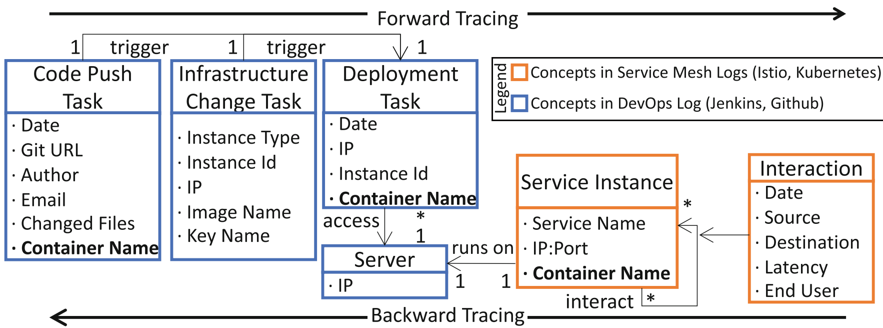


Fig. 3. The ontology alignment among DevOps and Service Mesh logs.

In DevOps logs are generated based on specific tasks such as code push and service deployment as stated earlier. We consider the code push task as pushing code to Github (a source code repository), containerizing the code to generate an image, and then pushing the image to Docker Hub (a container image repository). While pushing code to Github, the changed files and the committer's ID (email address) are recorded. The service deployment task is defined as accessing a server using a server IP via Secure Shell (SSH) and deploying a container image on a server. The logs of the code push task and the logs of service deployment task are aligned by a common attribute "Container Name".

The Service Mesh logs contain service instance information and interaction information among service instances. The service instance information contains a service name, an IP address, a port number and a container name. The interaction information contains an interaction date, a source service, a destination service, the latency of the interaction, and an end user who requested the interaction.

We align the ontologies (information) from DevOps and Service Mesh based on a common attribute "Container Name". By aligning the ontologies of DevOps and Service Mesh, we integrate service deployment and service interaction information. For example, if latency SLA is violated during interacting between two

services, the destination service IP is mapped onto the IP of a deployment task. Then, the image name of the deployment task is mapped onto the image name of the code push task. From the code push task, we can trace the email address of the code committer.

3.2 Proactive Change Detection and Notification

For assisting in fast communication and collaboration among the independent software teams regarding changes in services, Harmonia automates detecting changes in services, tracing the dependent services that can be impacted by the changes, notifying the changes to appropriate teams.

To determine what to detect and notify and whom to be notified, Harmonia follows predefined detection and notification (reference-) rules. Each rule consists of a detection condition (C) and a notification action (A). We define each rule as $C \rightarrow A$. Either the condition or the action can be specialized to customize the rules as ($C' \rightarrow A$) or ($C \rightarrow A'$)

Suppose Team A in Fig. 4, is responsible for the *Review service* and deploys the *Review service* after changing the code, Harmonia detects the deployment change from the deployment task logs and changed files from the code push task logs. We consider *Review service* as a *depender* service. We assume that there are services that might be impacted by the changes in the depender service. We refer such services as the *Dependee* services. The dependency can be extracted from recent service interaction logs of Service Mesh. For example, in the Fig. 4, the *Product Page service* that recently sent requests to *the Review service* is considered as the dependee service. By tracing the deployment information of the dependee (*Product Page service*), Harmonia notifies the changes of the depender (*Review service*) to the committer of the dependee (*Product Page service*).

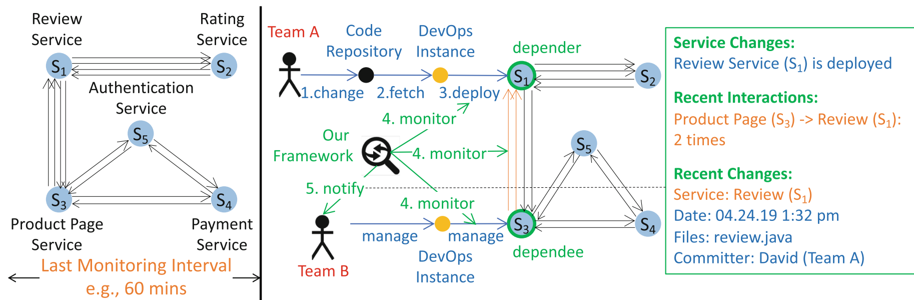


Fig. 4. Detecting changes and deployments of services and notifying to appropriate software teams.

We provide a reference rule (R_1) for the example scenario above as the following:

- **Condition** (C_1): The depender service (s) is deployed.

- **Action** (A_1): Notify the deployment and the change information of the depender service (s) to other teams responsible for the dependee services (d).

$$R_1 : deployed(s)_{C_1} \rightarrow \forall d \in dependee(s),\ notify(d, s)_{A_1} \tag{1}$$

Below, we refine the reference rule 1 (R_1) further. For example, if software teams responsible for the critical services (c), which are not direct dependees but had frequent interactions with the dependees, needs to be notified. Thus, we refine the rule as follows:

- **Condition** (C_1): The depender service (s) is deployed.
- **Action** (A_2): Notify the deployment and change information of the depender service (s) to teams responsible for the critical dependee services (c).

$$\begin{aligned} R_2 : deployed(s)_{C_1} &\rightarrow (\forall d \in dependee(s),\ notify(d, s)_{A_1}) \\ &\wedge (\forall c \in (\neg dependee(s) \wedge dependee(d) \wedge is_critical(s, c)),\ notify(c, s)_{A_2}) \end{aligned} \tag{2}$$

To implement the reference rules, we define (reference-) heuristic translation algorithms associated with the rules in Algorithm 1 and 2. The *deployed* procedure describes the steps of detecting changes and deployments of services by using DevOps logs and Service Mesh logs in a complementary manner. We assume that the logs are represented in the JSON format. The deployed process takes logs from code push, deployment and run-time interaction, notifies the change and deployment information to dependees, and then returns a list of deployments including change information. The deployed process shows a forward tracing from code push logs to run-time interaction logs to get the dependees of the newly deployed services.

The $notify$ procedure describes a backward tracing from service interaction logs to code push logs to extract contact email addresses of the dependee services. The notify process takes a list of dependees as notification targets and the deployment information of a depender service. Then it notifies the changes of the depender to dependees.

Transitive Impact Assessments. In addition to analyzing the impacts of changes on direct dependee services described earlier, Harmonia assesses two types of potential transitive impacts of service changes on other services. One is assessing impacts on a competitive service in using a common service as described in Fig. 5. The other one is assessing impacts on the other services invoked by the changed services implicitly as depicted in Fig. 6.

The difference between existing root-cause analysis approaches and our transitive impact assessment is the proactive change detection and notification. The root-cause analysis based approaches pinpoint the root-causes when multiple abnormal services are detected. In contrast, Harmonia considers newly changed services as root-causes and proactively infers the transitive impacts of the changes on other services.

Algorithm 1. A Heuristic Translation of *deployed* Condition to Code

```
1: pushedList ← read(pushed.json);        ▷ Obtained from Jenkins and Github Logs
2: deployedList ← read(deployed.json);                ▷ Obtained from Jenkins Logs
3: interactionList ← read(interaction.json);            ▷ Obtained from Istio Logs
4: namespaceList ← read(kubernetes.json);        ▷ Obtained from Kubernetes Logs
5: procedure DEPLOYED( )              ▷ Called regularly. Forward Tracing of Logs
6:     deployments, dependeeList, pushedList;
7:     for each deployed ∈ deployedList do▷ 1. Get recent changes and deployments
8:         for each pushed ∈ pushedList do
9:             if pushed.containerName == deployed.containerName then
10:                if pushed.date < deployed.date then
11:                    pushedList.add(pushed);
12:        deployed.put("changes", pushedList);
13:        for each namespace ∈ namespaceList do        ▷ 2. Get service names in
   production
14:            if deployed.containerName == namespace.containerName then
15:                deployed.put("serviceName", namespace.instanceName);
16:        for each interaction ∈ interactionList do             ▷ 3. Get dependees
17:            if interaction.destination == deployed.serviceName then
18:                dependeeList.add(interaction.source);
19:        deployed.put("dependees", dependeeList);
20:        deployments.put("deployments", deployed);
21:        notify(dependeeList, deployed);  ▷ 4. Notify the changes and deployments
   to dependees
22:    return deployments                                 ▷ A set of deployments
```

To assess the potential transitive impacts of service changes, we further define notations and rules for detecting and notifying service changes as below.

$$I = (\{S_{src}\}, \{S_{dst}\}, l, t), \quad S = (n, \{D\}), \quad D = (c_t, c_c, c_i) \tag{3}$$

The I is a set of individual interactions (i) within a time slot (T_{t-1}^{t}). Each interaction (i) consists of a source service (S_{src}), a destination service (S_{dstc}), an interaction latency (l), and a timestamp (t). Each service (S) is composed of a service name (n) and deployment information (D). The deployment information (D) involves a changed time (c_t), changed code information (c_c), and changed infrastructure information (c_i). The latency of a service interaction can be caused either by the changes in the source service or the changes in the destination service.

Figure 5 depicts a transitive impact assessment among competitive services. In this scenario, the service (S_3) and the service (S_5) are competing in invoking the common service (S_4), such as using a common API. A faulty change in the service (S_3) that occupies the service (S_4) with a longer period can impact the service (S_5). Harmonia detects the transitive relationship between competitive services by checking invoking sequences and latency propagations and then notifying service changes among the competitive services as described below:

Algorithm 2. A Heuristic Translation of *notify* Action to Code

```
1: procedure NOTIFY(tagets, deployed)                          ▷ Backward Tracing of Logs
2:   for each target ∈ targets do
3:     for each namespace ∈ namespaceList do        ▷ 1. Get container names of dependees
4:       if namespace.serviceName == target then
5:         target.put("containerName", namespace.containerName);
6:     for each pushed ∈ pushedList do              ▷ 2. Get contacts of the dependees
7:       if pushed.containerName == target.containerName then
8:         sendEmail(pushed.email, deployed);       ▷ 3. Send an Email with Deployment Information
```

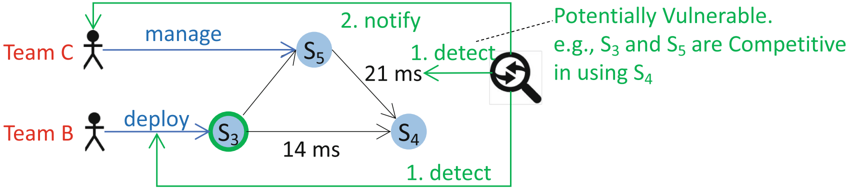


Fig. 5. A transitive impact assessment among competitive services.

- **Condition** (C_2): The interaction i_j (from S_j to S_m) and the interaction i_k (from S_k to S_m) have occurred within a time slot T^t_{t-1}. Service(S_j) and service (S_k) are competitive services that can impact each other in invoking the other service (S_m).
- **Condition** ($C_{1.1}$): Service (S_j) is deployed, which is an instance of C_1.
- **Condition** ($C_{1.2}$): Service (S_m) is deployed, which is an instance of C_1.
- **Action** ($A_{1.1}$): Notify the deployment and change information of service (S_j) to service (S_k) and service (S_m), which is an instance of A_1.
- **Action** ($A_{1.2}$): Notify the deployment and change information of service (S_m) to service (S_j) and service (S_k), which is an instance of A_1.

$$R_3 : (\forall i_j, i_k \in I^t_{t-1}, (i_j.S_{dst} = i_k.S_{dst}) \wedge (i_j.t < i_k.t) \wedge (i_k.l > i_j.l))_{C_2} \rightarrow$$
$$((i_j.S_{src}.D_{c_t} \in T^t_{t-1})_{C_{1.1}} \rightarrow (notify(i_j.S_{src}.D, i_j.S_{dst}) \wedge notify(i_j.S_{src}.D, i_k.S_{src}))_{A_{1.1}}) \vee$$
$$((i_j.S_{dst}.D_{c_t} \in T^t_{t-1})_{C_{1.2}} \rightarrow (notify(i_j.S_{dst}.D, i_j.S_{src}) \wedge notify(i_j.S_{dst}.D, i_k.S_{src}))_{A_{1.2}}) \quad (4)$$

Figure 6 shows a scenario of a transitive impact assessment for services invoking the other services implicitly. In this scenario, the service (S_3) is not a direct dependee of the service (S_2) but implicitly invokes the service (S_2). If a change in the service (S_2) increases the interaction latency between the service (S_1) and the service (S_2), the service (S_3) can be impacted. Harmonia captures the transitive impacts for services invoking other services implicitly by checking invoking sequences and latency propagations and then notifying service changes among the competitive services as described below:

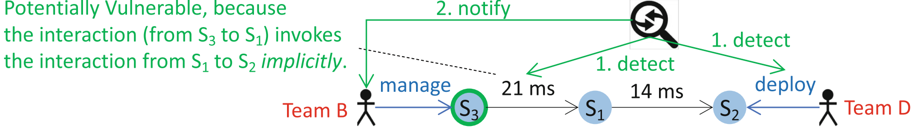


Fig. 6. A transitive impact assessment for services invoking the other services implicitly.

- **Condition** (C_3): The interaction i_j (from S_j to S_k) implicitly invokes the interaction i_k (from S_k to S_m).
- **Condition** ($C_{1.3}$): Service (S_m) is deployed, which is an instance of C_1.
- **Action** ($A^{'}_{1.1}$): Notify the deployment and change information of service(S_m) to service (S_j) and service (S_k), which is an instance of specialization of A_1.

$$R_4 : \forall i_j, i_k \in I^t_{t-1}, ((i_j.S_{dst} = i_k.S_{src}) \wedge (i_j.t < i_k.t))_{C_3} \wedge (i_k.S_{dst}.D \in T^t_{t-1})_{C_{1.3}} \rightarrow notify(i_k.S_{dst}.D, i_k.S_{src}) \wedge notify(i_k.S_{dst}.D, i_j.S_{src}))_{A^{'}_{1.1}} \quad (5)$$

The proactive detection and notification rules aim to provide forewarning among independent software teams. If the systems are sensitive for reliability, the forewarning would help the independent software teams in communicating and collaborating with richer information before an abnormality on the system is detected.

3.3 Reactive Change Detection and Notification

DevOps software teams are required to deliver services more frequently and independently to production environments, thereby increasing complexity in communication and collaboration among the teams. For example, if SLA violations occurred after deploying services independently, it is timing consuming to pinpoint causes of the SLA violations and impacted services and to notify the causes to appropriate teams.

Consider the Product page service experiences a 3 s delay after the Team A has deployed a new Review service as shown in Fig. 7. Harmonia automatically detects the violations of the latency SLA when the Product Page service sents a request to the Review service, tracking recent changes in the Review service, and notifying the recent changes to the teams responsible for the Product Page service to assist them react to the violation of latency SLA and remediate it.

A reference rule (R_3) for the example scenario above is defined as below:

- **Condition** (C_2): The latency of interaction from a source service (s_{src}) to a destination service (s_{dst}) is higher than a latency SLA (l_{SLA}).
- **Action** (A_2): Notify the recent changes in the destination service (s_{dst}) to the teams responsible for the service (s_{dst}) and dependee services (d).

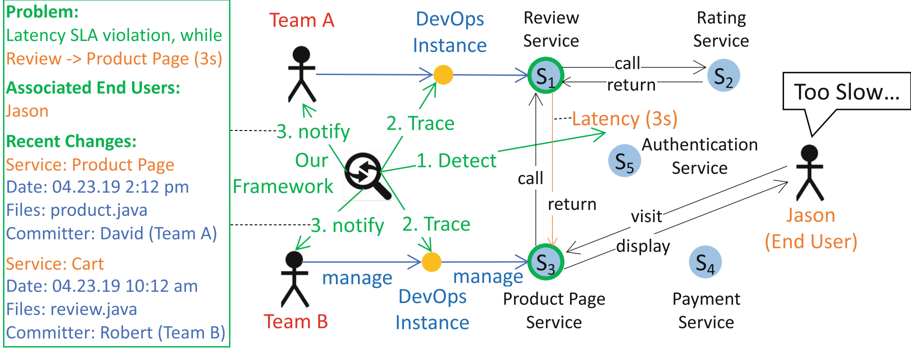


Fig. 7. Detecting SLA violations and impacted services and notifying recent changes in services to appropriate teams.

$$R_5 : \forall s_{dst} \in (latency(s_{src}, s_{dst}) > l_{SLA})_{C_2}) \rightarrow \\ notify(s_{dst}, s_{dst})_{A_2} \wedge (\forall d \in dependee(s_{dst}),\ notify(d, s_{dst})_{A_2}) \tag{6}$$

The latency of an interaction between two services can occur due to both changes in the source service and changes in the destination service. If dependees of the source service and the destination service need to be notified with the changes in source service and the destination service respectively, the reference rule 3 can be refined as below:

- **Condition** (C_2): The latency of interaction from a source service (s_{src}) to a destination service (s_{dst}) is higher than a latency SLA (l_{SLA}).
- **Action** (A'_2): Notify the recent changes in the source service (s_{src}) and destination service (s_{dst}) to the teams responsible for the dependees of source (d_{src}) and dependees of destination (d_{dst}).

$$R_6 : \forall s_{dst} \in (latency(s_{src}, s_{dst}) > l_{SLA})_{C_2}) \rightarrow \\ \forall d_{dst} \in dependee(s_{dst}),\ notify(d_{dst}, s_{dst})_{A_2}) \wedge notify(s_{dst}, s_{dst})_{A_2} \\ \wedge \forall d_{src} \in dependee(s_{src}),\ notify(d_{src}, s_{src})_{A'_2}) \wedge notify(s_{src}, s_{src})_{A'_2} \tag{7}$$

In Algorithm 3 for the rule R_4, the process for detecting violations of latency SLA and notifying appropriate teams, shows a backward tracing from run-time interaction logs to code push logs. The process assumes that the SLA specification is documented in the JSON. The process gets recent changes in the source and destination and then notify the changes to dependees of the source and destination.

Transitive Impact Assessments. In the transitive impact assessment phase, Harmonia pinpoints the root-causes of abnormal interactions, similar to the

Algorithm 3. A Heuristic Translation of *latency* Condition to Code

```
1: SLA ← read(SLA.json);
2: procedure LATENCY( )
3:     deployments ← depolyed();
4:     for each interaction ∈ interactionList do
5:         if interaction.latency > SLA.latency then
6:             for each namespace ∈ namespaceList do
7:                 if deployment.serviceName == interaction.destination then
8:                     notify(interaction.destination, deployment);
9:                     notify(interaction.destination.dependees, deployment);
10:                if deployment.serviceName == interaction.source then
11:                    notify(interaction.source, deployment);
12:                    notify(interaction.source.dependees, deployment);
```

existing root-cause analysis based approaches. However, Harmonia goes beyond by providing richer information to software teams with an understanding of potential reasons why such abnormal interactions occurred. Harmonia additionally pinpoints and notifies recent code changes and infrastructure changes in abnormal services as a starting point of inspection, towards facilitating communication and collaborations among independent teams and fixing the issues more quickly.

For assessing the actual impacts among competitive services, Harmonia utilizes the rule (R_3) defined during the proactive detection and the notification phase. In the scenario described in Fig. 5, if the latency (l) of the interaction (i_j) (from service S_3 to service S_4) violates the latency SLA (l_{SLA}), Harmonia detects the code or infra changes and notifies to impacted teams. The detection and notification rules are specialized from the rule (R_3) and defined as below:

- **Condition** ($C_{2.1}$): The latency of interaction from a source service (S_{src}) to a destination service (S_{dst}) is higher than a latency SLA (l_{SLA}) in an interaction (i_j).
- **Condition** ($C_{2.2}$): The latency of interaction from a source service (S_{src}) to a destination service (S_{dst}) is higher than a latency SLA (l_{SLA}) in an interaction (i_k).

$$R_7 : \forall i_j, i_k \in I_{t-1}^{t}, C_2 \wedge (i_j.l > l_{SLA})_{C_{2.1}} \wedge (i_k.l > l_{SLA})_{C_{2.2}} \rightarrow A_{1.1} \wedge A_{1.2} \quad (8)$$

Similarly, Harmonia utilizes the rule (R_4) for the detection and notification rule for services invoking the other services implicitly. In the scenario described in Fig. 6, if the interaction between the service (S_1) and the service (S_2) violates the latency SLA due to a change in service (S_2), Harmonia detects the latency violations and notifies the change to impacted teams as defined below:

$$R_8 : \forall i_j, i_k \in I_{t-1}^{t}, C_3 \wedge C_{1.3} \wedge (i_j.l > l_{SLA})_{C_{2.1}} \wedge (i_k.l > l_{SLA})_{C_{2.2}} \rightarrow A_{1.1} \quad (9)$$

4 Harmonia in Action

To assist with an initial understanding of the applicability of Harmonia, we applied Harmonia to the Bookinfo application[2], which is adopted as an official sample application to describe the Istio framework, and compared the information collected from Harmonia with the information obtained from existing frameworks. The Bookinfo application displays the information of a book, including a description of the book, book details (ISBN, number of pages, etc.), and book reviews. The Bookinfo composed of four separate microservices - Product Page, Detail, Review, and Rating. Jenkins is adopted in our demonstration to build a sample DevOps pipeline, which consists of jobs for pushing code to Github, building and containerizing the code, and deploying the container.

4.1 Experimentation Setting

Four Github accounts are assigned to the Review service, the Detail service, and the Rating service. Each account is considered as a contact point of a software team that is responsible for a service. A total of 1200 visitors to the Bookinfo application are simulated. Sixty visitors per second are generated and access the Bookinfo application through a gateway service. Two types of Rating service are alternatively deployed every 10 s. We injected faulty code for causing delays from one second to seven seconds in communicating with the review service to one of the rating services. The other rating service works without causing delays. Harmonia collected the service deployment information and service interaction information every 10 s, detected violations of latency SLA, source of changes, and impacted services, notified the violations and changes to the four contacts. The latency SLA is given as one second. Then, we collected and compared the information from Harmonia, Github, Jenkins, Kubernetes, and Istio.

4.2 Observation and Discussion

Table 1 summarizes the experimentation results, showing a quantitative comparison with existing frameworks that measure the types of available information regarding service changes.

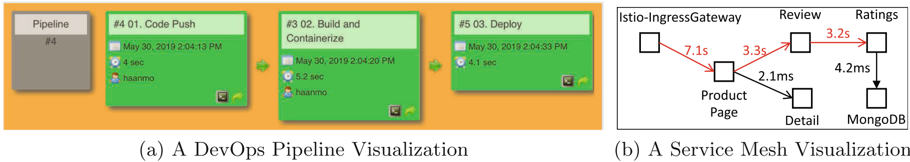
The Github logs captured the 20 times of code changes, including lines of changed code. The Jenkins logs captured the 20 times of code commitment history, containerization history of the code, and deployment history of the container. However, both Github and Jenkins have a lack of service run-time information after the deployments. The Kubernetes logs captured the 20 times of service container deployment history, and the Istio logs captured the number of visitors to the Bookinfo application, the total number of interactions among services, and the latency of the interactions. However, the Kubernetes and Istio do not capture the information of changes in the service containers that are newly deployed. Harmonia bridged the dichotomy between the DevOps tools

[2] https://istio.io/docs/examples/bookinfo/.

Table 1. A quantitative comparison with existing frameworks.

	Code changes	Service deployments	Problematic deployments	Visitors	Service interac-tions	SLA violations	Identified root-cause	Identified impact
Total # of Changes	20	20	10	1,200	10,753	932	220	712
Harmonia	20	20	10	1,200	10,753	932	220	712
Github	20	-	-	-	-	-	-	-
Jenkins	20	20	-	-	-	-	-	-
Kubernetes	-	20	-	-	-	-	-	-
Istio	-	-	-	1,200	10,752	932	-	-

and Service Meshes by extracting and aligning the logs from the tools. Additionally, based on the logs, Harmonia deduced ten problematic service deployments, 220 root-cause interactions that caused by problematic deployments, and 712 impacted interactions and notified the source of code changes and deployments that cause the violations. To evaluate whether root-cause of changes can be identified, refer to Fig. 6, we injected a faulty code in the rating service (S_2) (i.e., the root cause) which introduces delays in the interaction between the review service (S_1) and the rating service (S_2). The interaction impacts on the other interaction between the product page service(S_3) and the review service (S_1). Among the total of 932 abnormal interactions that violate the latency SLA, Harmonia detects the 220 interactions between the rating (S_2) and review (S_1) as root-cause interactions. The 712 impacted interactions represent the interactions between the product page (S_3) and the review page (S_1).



(a) A DevOps Pipeline Visualization (b) A Service Mesh Visualization

Fig. 8. AS-IS visualizations of a DevOps pipeline and a service mesh

Figures 8 and 9 show a visual comparison between Istio, Jenkins, and Harmonia. The Jenkins pipeline described in Fig. 8a contains service deployment information, such as deployment date, code changes, etc., but rarely involves run-time service information after the deployments. On the other hands, as depicted in Fig. 8b, Istio visualizes the run-time interactions and latencies among services but has a lack of understanding about what kinds of service changes cause the latency variations.

As described in Fig. 9, Harmonia visualizes the service deployment information and run-time interaction information in a single view for helping independent software teams in understanding the impact of changes in services.

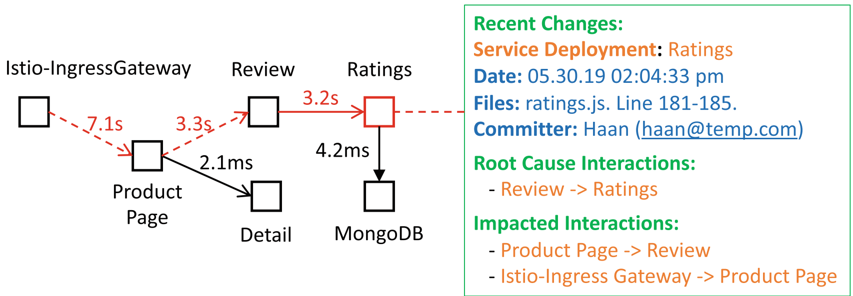


Fig. 9. A visualization of harmonia prototype

The Rating is colored red as the deployments of the Rating violated the latency SLA. The interaction between the Review and the Rating colored red with a solid line as it is a root-case interaction. The impacted interactions colored red with dotted lines.

4.3 Threats to Validity

Currently, as Harmonia understands an integrated body of knowledge from specific logs of Github, Jenkins, Istio, and Kubernetes, the Harmonia ontology is limited to be generalized. The Harmonia reference rule set for detection and notification is limited and straightforward yet to apply Harmonia to more diverse domains. In addition, Harmonia utilizes a centralized point of a knowledge base, whereas microservices build on independent teams with separation of concerns. It would be necessary to decentralize the knowledge base appropriately in terms of access control, ownership, and trust.

5 Conclusion

In this paper, we presented Harmonia - a continuous service monitoring framework using both DevOps and Service Mesh in a complementary manner, as among the first of its kind to the best of our knowledge, to facilitate communication and collaborations among DevOps software teams independently contributing to service changes. Harmonia offers a reference ontology alignment of DevOps logs and Service Mesh logs to have an integrated body of knowledge between service deployment information from DevOps that includes code and infrastructure changes of services and run-time service information from Service Mesh that captures run-time interactions among service along with its latency. Harmonia also offers detection and notification rules to enhance the understandability of the changes in services and impacts of the changes.

To generalize our approach, we are planning to expand the ontology to cover other DevOps and Service Mesh frameworks, such as Puppet, Chef, or Linke rd. To enhance the Harmonia reference rule set, we are also planning to consider

more complex cases based on studying real application services in cloud-native production environments. We would also like to further evaluate to what extent can Harmonia notifications help development teams performing change management and diagnosis.

References

1. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
2. Brunnert, A., et al.: Performance-oriented DevOps: a research agenda. CoRR abs/1508.04752 (2015). http://arxiv.org/abs/1508.04752
3. Chen, P., Qi, Y., Hou, D.: CauseInfer: automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. IEEE Trans. Serv. Comput. **12**(2), 214–230 (2019)
4. Fadda, E., Plebani, P., Vitali, M.: Monitoring-aware optimal deployment for applications based on microservices. Trans. Serv. Comput. 1–1 (2019)
5. Fitzgerald, B., Stol, K.J.: Continuous software engineering and beyond: trends and challenges. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, pp. 1–9. ACM, Hyderabad (2014)
6. Forsgren, N., Kim, G., Kersten, N., Humble, J., Brown, A.: 2017 state of devops report. Puppet+ DORA
7. Gupta, M., Mandal, A., Dasgupta, G., Serebrenik, A.: Runtime monitoring in continuous deployment by differencing execution behavior model. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) ICSOC 2018. LNCS, vol. 11236, pp. 812–827. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03596-9_58
8. Haselböck, S., Weinreich, R.: Decision guidance models for microservice monitoring. In: Proceedings of the International Conference on Software Architecture Workshops (ICSAW), pp. 54–61. IEEE (2017)
9. Heinrich, R., et al.: Performance engineering for microservices: research challenges and directions. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, pp. 223–226. ACM, L'Aquila (2017)
10. Jayathilaka, H., Krintz, C., Wolski, R.: Performance monitoring and root cause analysis for cloud-hosted web applications. In: Proceedings of the 26th International Conference on World Wide Web, pp. 469–478. International World Wide Web Conferences Steering Committee, Perth (2017)
11. Johng, H., Kim, D., Hill, T., Chung, L.: Estimating the performance of cloud-based systems using benchmarking and simulation in a complementary manner. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) ICSOC 2018. LNCS, vol. 11236, pp. 576–591. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03596-9_42
12. Johng, H., Kim, D., Hill, T., Chung, L.: Using blockchain to enhance the trustworthiness of business processes: a goal-oriented approach. In: 2018 IEEE International Conference on Services Computing (SCC), pp. 249–252. IEEE (2018)
13. Kalia, A.K., Xiao, J., Bulut, M.F., Vukovic, M., Anerousis, N.: Cataloger: catalog recommendation service for IT change requests. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 545–560. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_40
14. Kim, M., Sumbaly, R., Shah, S.: Root cause detection in a service-oriented architecture. In: Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems, pp. 93–104. ACM, Pittsburgh (2013)

15. Len Bass, I.W., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Old Tappan (2015)
16. Lin, J., Chen, P., Zheng, Z.: Microscope: pinpoint performance issues with causal graphs in micro-service environments. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) ICSOC 2018. LNCS, vol. 11236, pp. 3–20. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03596-9_1
17. Motahari, H., Benatallah, B., Saint-Paul, R., Casati, F., Andritsos, P.: Process spaceship: discovering and exploring process views from event logs in data spaces. Proc. VLDB Endow. **1**(2), 1412–1415 (2008)
18. Phipathananunth, C., Bunyakiati, P.: Synthetic runtime monitoring of microservices software architecture. In: Proceedings of 42nd Annual Computer Software and Applications Conference (COMPSAC), vol. 02, pp. 448–453 (2018)
19. Pina, F., Correia, J., Filipe, R., Araujo, F., Cardroom, J.: Nonintrusive monitoring of microservice-based systems. In: Proceedings of the 17th International Symposium on Network Computing and Applications (NCA), pp. 1–8. IEEE (2018)
20. Wang, P., et al.: Cloudranger: root cause identification for cloud native systems. In: Proceedings of 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 492–502 (2018)
21. Xiao, J., Kalia, A.K., Vukovic, M.: Juno: an intelligent chat service for IT service automation. In: Liu, X., et al. (eds.) ICSOC 2018. LNCS, vol. 11434, pp. 486–490. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17642-6_49
22. Zhu, L., Bass, L., Champlin-Scharff, G.: Devops and its practices. IEEE Softw. **33**(03), 32–34 (2016)