A fault injection platform for learning AIOps models

Frank Bagehorn
fba@zurich.ibm.com
IBM Research GmbH
Switzerland

Jesus Rios
jriosal@us.ibm.com
T.J. Watson Research Center, IBM
United States

Saurabh Jha
Saurabh.Jha@ibm.com
T.J. Watson Research Center, IBM
United States

Robert Filepp
filepp@us.ibm.com
T.J. Watson Research Center, IBM
United States

Naoki Abe
nabe@us.ibm.com
T.J. Watson Research Center, IBM
United States

Laura Shwartz
lshwart@us.ibm.com
T.J. Watson Research Center, IBM
United States

Xi Yang
xi.yang@ibm.com
T.J. Watson Research Center, IBM
United States

ABSTRACT

In today's IT environment with a growing number of costly outages, increasing complexity of the systems, and availability of massive operational data, there is a strengthening demand to effectively leverage Artificial Intelligence and Machine Learning (AI/ML) towards enhanced resiliency. In this paper, we present an automatic fault injection platform to enable and optimize the generation of data needed for building AI/ML models to support modern IT operations. The merits of our platform include the ease of use, the possibility to orchestrate complex fault scenarios and to optimize the data generation for the modeling task at hand. Specifically, we designed a fault injection service that (i) combines fault injection with data collection in a unified framework, (ii) supports hybrid and multi-cloud environments, and (iii) does not require programming skills for its use. Our current implementation covers the most common fault types both at the application and infrastructure levels. The platform also includes some AI capabilities. In particular, we demonstrate the interventional causal learning capability currently available in our platform. We show how our system is able to learn a model of error propagation in a micro-service application in a cloud environment (when the communication graph among micro-services is unknown and only logs are available) for use in subsequent applications such as fault localization.

KEYWORDS

Fault injection, Fault diagnosis, AI supported operations

ACM Reference Format:

Frank Bagehorn, Jesus Rios, Saurabh Jha, Robert Filepp, Naoki Abe, Laura Shwartz, and Xi Yang. 2022. A fault injection platform for learning AIOps models. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559503

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559503

1 INTRODUCTION

The Uptime Institute's 2022 Outage Analysis Report [19] states that the number of publicly reported outages that last longer than 24 hours increased from 8% to 28% over the last 5 years. Cost analysis of outages also shows a sharp increase from 2019 to 2021: outages with an estimated cost between $100,000 and $1 million increased from 28% to 47%, and outages costing over $1 million grew from 11% to 15% percent. With third-party commercial operators such as cloud hosting and collocation providers accounting for the majority 71% (in 2021) of outages, outages with root causes in operations top the list. While significant efforts have been made on improving bug localization techniques, the research focusing on reducing operational issues for critical production applications is still limited.

Contemporary micro-services architectures simplified the scope of software developers, but the roles of Ops/Software Reliability Engineers (SREs) have become even more complex. Today an IT environment with a large number of inter-dependent components, using virtualization at every level, can generate millions of transactions a day and its components can change every few seconds. The vast volume of observability data and dynamism of the IT environments make it difficult to assess the resiliency posture of an application. It also causes longer issue resolution time in production. In addition, software releases have become more frequent due to widely adopted agile development, limiting applicability of prior knowledge that Ops/SREs have and increasing the complexity of their role.

Infusing Artificial Intelligence and Machine Learning (AI/ML) into Ops tools is one of the responses to the ever increasing complexity of Ops/SREs that is being adopted by many enterprises today. However, this approach heavily relies on historical data which could have limited applicability as described above. Our automated fault injection-based learning solution described in this paper was built by an IBM Research team to alleviate the issues that AIOps experience with stale, incomplete, and potentially misleading historical data.

Our solution is a fault injection service that allows the simulation of user-defined faults into an application. The use of a fault injection service in a staging environment enables the training of supervised machine learning models using data collected during fault injection simulations. These models can then be used in real time for the localization of actual faults and the discovery of performance issues in production. It is worth emphasizing that, by coupling the data collection and machine learning steps, our solution makes it possible to *optimize* the collection of data for the learning task at hand. Also performing them in a staging environment minimizes the overhead due to data collection and learning during operations.

Fault injection is a common practice used in chaos engineering or application testing. While Chaos Engineering is used in production environments, we position our solution as an automated tool operating in staging or testing environments targeting Operations space. Our internal teams use the solution a) to learn about their applications' operational behaviour in a faulty state, b) to generate related operational data that can be used for training and validation of AIOps models. The models, with subsequently limited calibration, are crucial in helping Ops/SREs in preventing, predicting, and resolving disrupting and costly outages.

2 OBSERVATIONS AND CHALLENGES

There is a growing need to support feedback-driven and MAPE-based[1] fault injection technologies to develop, train and test the AIOps models. Training and verification of AIOps models as well as other machine learning tasks are performed mostly by data scientists who are used to feedback-driven methodology. MAPE also allows for pruning of the fault space by learning previous fault injections. To ensure ease of use, a fault injection framework must also provide inbuilt data collection and observability.

Our experience shows that the injection of faults (also referred to here as interventions) in production environments is not widely accepted due to possible adverse effects. Some interventions can turn out to be costly or even dangerous [10], and therefore, conducting fault injection in production environments is highly undesirable. However, while observations of an application in production can provide some information about the statistical relations among events, only interventions can generate data that enables us to differentiate among the different causal structures. To take advantage of interventional learning we use the fault injection framework in *staging environments.*

Contemporary cloud applications are commonly built upon fine-grained modularity, consisting of multiple single-purpose and loosely-coupled *micro-services*, which are containerized and executed on platforms like Kubernetes® clusters. Micro-service-based applications are deployed in cloud-native, hybrid, and multi-cloud environments. In fact, only 20% of the applications are fully deployed in the cloud-native environment [1]; thereby limiting the use of existing fault injection and chaos toolkits focused on containerized applications.

Data scientists tasked with creation and maintenance of AI Ops models have skill sets which are different from software engineers, application developers and SREs. Their programming skills may be limited to certain programming languages or libraries, and their knowledge of modern infrastructure could be limited. However, many existing chaos engineering tools assume a great deal of development skills and deep understanding of compute environments. Let us illustrate that by looking at some of the most widely used chaos engineering tools.

- Chaos toolkit [2] supports a long list of different fault types for different environments, thus supporting hybrid clouds. It has spawned an ecosystem, where additional fault types for specific platforms are added by third-party developers. But it is provided as a library that needs glue code to create chaos experiments.
- Litmus [11], a cloud native chaos engineering platform, provides a user interface (UI) to create chaos experiments and supports workflows to orchestrate a set of those. However, it still requires a good understanding of Kubernetes® and its YAML configuration files, as one might have to deal with service account configuration and role-based privileges to get the experiments to run.
- Gremlin [5] supports chaos engineering for a variety of operating systems and platforms, whether these are container platforms, virtual machines, or bare metal servers. Its server-client architecture requires an agent to be installed on each target system. Attacks are then configured on the Gremlin control-plane and executed by the agent the next time it contacts the control-plane.

In order to support the use of fault injection in the data science community, one must move away from this programming oriented paradigm and instead focus on

- Feedback loop: The number of possible fault injection points grows with the number of application components. Together with a potentially large parameter space for each fault, a feedback loop helps to dynamically adjust the test plan and reduce the number of tests to an amount feasible in practice.
- Ease of use: Fault injection scenarios should be constructed and executed in an intuitive way, without any kind of programming or deep application platform knowledge.
- Orchestration: Real world complex fault scenarios often can be reduced to a set of simple fault types, that act together in an orchestrated way.
- Minimal intrusion: As the application can span multiple systems, clusters or clouds, no installation tasks should be required on those to execute the fault injection.

2.1 Example use-case: Learning failure propagation and application communication graph

In this paper, we focus on showcasing the power of our solution by developing an AIOps model that allows us to learn the communication graph for a micro-service-based application that may execute in a cloud-native, hybrid or multi-cloud environment. We address the problem of learning the communication graph for a micro-service-based application because the communication graph is not easily available to the platform-provider in hybrid or multi-cloud environment. The knowledge of the *communication graph* among micro-services is beneficial to the platform provider as the

[1]MAPE: Monitor, Analyze, Plan and Execute.

graph can be used for training a variety of AIOps models such as optimizing the management and many downstream tasks like active probing [18], testing [7], performance diagnosis and mitigation [14], taint analysis [3] and fault localization [20].

In machine learning terms, learning the communication graph can be modeled as an *interventional causal learning* problem to infer the possible communication links that exist between various micro-services in the system. More specifically, it can be learned via interventions by *fault injection*: when programmatically injecting a fault over a certain micro-service, other dependent micro-services calling it will receive an error and in turn throw exceptions. By capturing such fault propagation among the micro-services, it is possible to infer the communication graph and use it for proper fault management in production.

3 OUR APPROACH

To tackle the challenges and observations described in Sec. 2 we have developed a fault injection platform that supports AI learning tasks. The architectural principles that we followed are:

- Combine fault injection and data collection in a single framework, so learning of AI models does not need to manually collect the operational data about the application from different sources
- Support hybrid cloud or multi-cloud environments
- Make fault injection accessible to non-programmers (e.g., Q&A testers, data analysts or data scientists) thus potentially reducing the training and/or operational cost
- Zero or minimal footprint on the environments under test

As an example of an AI model created within our platform, in Sec. 4 we showcase how it enables a causal learning environment to learn the communication or error propagation graph of an application.

The platform consists of a number of different micro-services that are deployed in a Kubernetes® cluster (see Figure 1). It has been implemented as a set of FastAPI [15] micro-services written in Python™.

The *fault injection micro-service* provides API endpoints for various operations. It allows to insert/replace/patch or delete authentication configurations, supporting the injection of faults in remote Kubernetes® or Red Hat® OpenShift® [16] clusters as well as virtual machines or bare metal servers running a flavor of Linux®. Other supported operations are the injection or removal of faults, the listing of currently injected faults and their specs, the listing of a history of faults injected into the various targets, or the listing of all supported fault types along with the description of required parameters and target resource type compatibility.

For the fault injection itself, the micro-service calls some logic from libraries. This can be logic, like the one we implemented ourselves for some infrastructure faults, but it could as well be logic from a third party library like chaos toolkit [2]. This way, specific faults become available to the end user through simple API calls. The richness of fault types available through existing fault injection ecosystems allows for the integration of faults targeting various platforms and layers covering the whole stack from infrastructure and network to middle ware and the application layer.

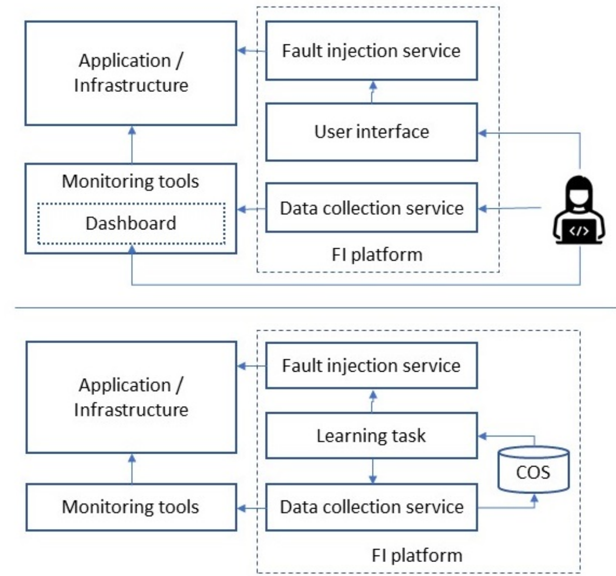

Figure 1: Architecture of the Fault injection (FI) platform: manual exploration vs. automated learning

The current implementation covers faults from four categories: Kubernetes® resource errors, HTTP errors in Istio® enabled applications, saturation faults, and network errors. See Table 1 for a list of fault types in each category. We have conducted experiments with Istio's east-west gateway to extend the service mesh to VMs or bare metal servers running Linux, but the required installation steps make it impractical to use in a customer environment. As we continue the development of the platform, we will cover more platforms, e.g. IBM Z, as well as more fault categories, e.g. by utilizing the rich ecosystem of the chaos toolkit [2].

The fault injection API is called with a list of faults as input. The faults in the list are described through a simple JSON dictionary, where one specifies the target resource, a fault type, the authentication context, and other required information like fault specific parameters. To construct more complex fault scenarios, the service allows for simple fault orchestration by specifying a start delay and/or fault duration time. For example, a growing memory consumption could be emulated by a series of memory consumption faults with increasing memory values. Some typical fault combinations have been pre-defined and are readily available for injection.

To have an even further simplified end-user experience for data exploration, another micro-service provides an *user interface* (UI) (see Figure 1). Once again, the end-user would specify some connection configuration and credentials. With these credentials the UI connects to a Kubernetes® cluster, for example, to collect information about existing services or pods in a given namespace, so when the user picks a fault type from a list, available compatible targets are listed automatically and can easily be selected. Finally it makes calls to the fault injection micro-service API to execute the fault injection.

Table 1: Supported fault types by category

Fault category	Fault type
K8s resource errors *(Kubernetes® only)*	Kill pods, containers Drain/cordon nodes
HTTP errors *(Istio service mesh only)*	HTTP delay HTTP timeout HTTP abort (error response code)
Saturation *(Supports Hybrid Cloud)*	CPU hog Memory hog I/O load Network load HTTP endpoint load
Network errors *(Supports Hybrid Cloud)*	Packet delay Packet reordering Packet duplication Packet corruption Packet loss Bandwidth restriction

A *data collection micro-service* complements the fault injection service. It supports the collection of operational data like alerts, events, logs, traces, etc. from external operational monitoring systems like Mezmo (former LogDNA [12]), Instana [6] and others. The provided API calls hide the syntax and the setting of required parameters of the external system API calls from the end-user and allow for both the manual exploration as well as the automated collection and integration of operational data from different sources for learning tasks.

To fully automate the cycle of fault injection, data collection, and model learning, the data collection service can store the data in a Cloud Object Store (COS) bucket, where it is easily picked up by a learning algorithm (see Figure 1).

4 EXPERIENCE IN LEARNING FAILURE PROPAGATION MODELS

Modern cloud applications are commonly built using micro-service architectures, due to their ability to increase the speed and frequency of software delivery. In these applications, user requests trigger sequences of calls between micro-services. Obtaining a map of how micro-services communicate with each other is not always easy. However, knowledge of the communication graph could be very useful to have for tasks like active probing, testing, and fault localization.

For example, when a micro-service fails, micro-services calling it receive an error which may propagate all the way to the system user if not handled properly by the affected micro-services. Thus, a fault in one micro-service can generate a cascade of failures in many other micro-services. An SRE observing all these errors from multiple locations in the system may have a hard time trying to find out where the problem originated. The communication graph of the application in this case would be very useful in reasoning about the root cause.

For this use case, we have used the fault injection platform described in Sec. 3 in a staging environment to generate the fault data needed to learn a model of error propagation for any micro-service application when observability is limited to logs data. Kubernetes® is a container management platform widely used in cloud computing. While it provides tracing capabilities to capture message flows between micro-services, here we address the case in which only logs are available because the great majority of production applications are not yet managed by Kubernetes and may not be easily traced. In fact, 80 percent of respondents to a recent survey conducted by Canonical [1] reported that most, or all, of their applications are not yet managed by Kubernetes. Absence of tracing capability is also typical in hybrid-cloud environment which is common in enterprise applications [8].

Learning how errors propagate will provide a model of *causal relations* between errors occurring at micro-services (i.e. whether an error in micro-service A is caused by an error in micro-service B). We can then leverage this causal knowledge learned in a staging environment to perform fault localization (i.e. identifying the originating micro-service with the fault) in production by analyzing observed error logs.

To learn such a model in a staging environment, we automatically and sequentially inject an availability fault in each micro-service of the application using the fault injection framework. Under a load that simulates typical user requests to the application, we collect all the logs coming out of each micro-service during fault injection. Once all the data have been collected we apply causal learning to estimate the error propagation model by converting the log data into multiple time series, one per micro-service, counting the number of error logs per time interval. The specific algorithm we use is one that is adopted from the interventional causal learning literature [9], and describing the algorithmic details is outside the scope of this paper. We note that typically, the error propagation model, the causal graph of interest, is the reverse of the communication graph as errors propagate in a direction reverse to the direction of communication when they are not properly handled.

Fault injection in a staging environment provides us with the data necessary to learn how errors propagate in a micro-service application before deployment. This information is useful to help localize faults during production via root cause analysis of errors. We note that the type of data generated by our fault injection platform is not necessarily readily available during production, both because of limited observability and lack of ground truth knowledge on the root causes.

We have validated our system using two publicly available benchmark micro-service applications (DayTrader [4], a small application with 5 services, and TrainTicket [13] with 33 services) to learn their communication graph and error propagation model. To evaluate the learned causal error propagation patterns, we calculate the evaluation metrics of precision, recall, and F1-score, by comparing our induced error propagation graph versus the ground truth graph, as shown in Table 2. Specifically, precision indicates what proportion of identified edges actually have connections; recall tells what proportion of edges that actually have connections can be correctly detected by the algorithm; F1-score is a harmonic mean of precision and recall that sets their trade-off. The results show the algorithm can effectively learn the causal error propagation

patterns, achieving the F1-scores of 1.00 for DayTrader and 0.62 for TrainTicket. When the number of micro-services is relatively small, e.g., DayTrader, the causal relationship can be fully captured by the algorithm; As the number of micro-services grows, e.g., TrainTicket, it becomes much more challenging.

Table 2: Fault injection causal learning results evaluated against the ground truth for the DayTrader and TrainTicket micro-service benchmark applications

Application	Precision	Recall	F1-score
DayTrader	1.00	1.00	1.00
TrainTicket	0.73	0.54	0.62

Once an error propagation model is learned, the AI team was able to test a fault localization algorithm that consumes such a model and compare its performance with current fault localization services in our products. The fault injection platform was used for this evaluation by injecting faults, one at a time, in the benchmark applications and checking whether the injected faults were localized correctly. Each fault localization estimate produced by the algorithm once a fault is detected consisted of a set of candidate micro-services where it is believed the fault may be located. Table 3 shows the experimental results from the evaluated fault localization algorithm. Specifically, the fault localization accuracy (as the percentage of injected faults that were correctly localized, i.e., the location of the injected fault was in the estimation set), average size of the estimated location sets, and the corresponding informativeness[2] (with a value of 1 if the estimation set consists of only one location and 0 for the case where the estimation set is as large as the total number of micro-services in the application) are reported.

Table 3: Fault localization experimental evaluation

Application	Accuracy	Avg. set size	Informativeness
DayTrader	1.00	1.20	0.95
TrainTicket	0.94	5.31	0.87

5 CONCLUSION

We have shown how an automatic fault injection platform enables and optimizes the generation of data needed for building AI/ML models to support modern IT operations. The demonstrated framework supports a variety of operating environments, hybrid and multi-cloud as well as virtual machines and bare metal servers.

Motivated by the needs of data scientists and AI researchers we combine fault injection and data collection to support automated learning algorithms that create models that can effectively support operational tasks in production systems. We also demonstrated the effectiveness of the platform with a concrete use case in which the error propagation graph of a micro-service based application is learned using one of the AI/ML capabilities we have implemented (interventional causal learning). The learned graph was used in a downstream application for fault localization, which, in turn, its usefulness was evaluated using again our automated fault injection platform.

Our solution is in use by IBM consulting and deployed by the IBM Sales Cloud team. We continue enhancing it based on the feedback we received from our users: expanding supported operating environments, extending the number of fault categories and fault types and adding more data collection sources.

In the future, we plan to investigate the use and needs of the fault injection framework in a broker-enabled multi-cloud environment (sky) [17], that abstracts out the selection of cloud providers from application-centric roles like data scientist, developer, SRE, etc.

[2] Informativeness basically represents the percentage of micro-services that are not in the fault location estimated set. Thus, the more exclusions in the set the more informative the set estimate.

REFERENCES

[1] Canonical Ltd. . 2022. *Kubernetes and cloud native operations report 2022.* Retrieved June 22, 2022 from https://assets.ubuntu.com/v1/ee0365d8-Kubernetes+cloud+native+operations+report+2022_10.05.22.pdf
[2] Chaos Toolkit team. 2017. *The Chaos Engineering toolkit for Developers.* Retrieved June 15, 2022 from https://chaostoolkit.org/
[3] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis.* 196–206.
[4] DayTrader . 2022. . Retrieved June 15, 2022 from https://github.ibm.com/ocp-r2-demo/
[5] Gremlin Inc. 2022. *Proactively improve reliability.* Retrieved June 15, 2022 from https://www.gremlin.com/
[6] Instana Inc. 2022. *Enterprise Observability and APM for Cloud-Native Applications.* Retrieved June 20, 2022 from https://www.instana.com
[7] Saurabh Jha, Subho Banerjee, Timothy Tsai, Siva KS Hari, Michael B Sullivan, Zbigniew T Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. 2019. Ml-based fault injection for autonomous vehicles: A case for bayesian fault injection. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN).* IEEE, 112–124.
[8] Denis Kennelly. 2019. *Three reasons most companies are only 20 percent to cloud transformation.* Retrieved June 19, 2022 from https://www.ibm.com/blogs/cloud-computing/2019/03/05/20-percent-cloud-transformation/
[9] Murat Kocaoglu, Karthikeyan Shanmugam, and Elias Bareinboim. 2017. Experimental design for learning causal graphs with latent variables. In *Nips.*
[10] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643* (2020).
[11] Litmus. 2020. *Cloud Native Chaos Engineering platform.* Retrieved June 15, 2022 from https://litmuschaos.io/
[12] Mezmo Inc. 2022. *Log Analysis and Log Management Software for Observability Data.* Retrieved June 20, 2022 from https://www.mezmo.com/
[13] Software Engineering Laboratory of Fudan University. 2018. *Train Ticket: A Benchmark Microservice System.* https://github.com/FudanSELab/train-ticket/
[14] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for {SLO-Oriented} Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* 805–825.
[15] Sebastián Ramírez. 2022. *FastAPI.* Retrieved June 20, 2022 from https://fastapi.tiangolo.com/
[16] Red Hat®. 2022. *Red Hat OpenShift.* Retrieved June 17, 2022 from https://www.redhat.com/en/technologies/cloud-computing/openshift
[17] Ian Stoica and Shenker Scott. 2021. From Cloud Computing to Sky Computing. In *HotOS '21: Proceedings of the Workshop on Hot Topics in Operating Systems.* ACM, 26–32.
[18] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. {NetBouncer}: Active Device and Link Failure Localization in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* 599–614.
[19] Uptime Institute. 2022. *Uptime2022.* Retrieved June 19, 2022 from https://uptimeinstitute.com/webinars/webinar-critical-update-uptime-institute-2022-outage-report
[20] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 683–694.