Delivering software with agility and quality in a cloud environment

F. Oliveira
T. Eilam
P. Nagpurkar
C. Isci
M. Kalantar
W. Segmuller
E. Snible

Cloud computing and the DevOps movement are two pillars that facilitate software delivery with extreme agility. "Born on the cloud" companies, such as Netflix®, have demonstrated rapid growth to their business and continuous improvement to the service they provide, by reportedly applying DevOps principles. In this paper, we claim that to fulfill the vision of fast software delivery, without compromising the quality of the provided services, we need a new approach to detecting problems, including problems that may have occurred during the continuous deployment cycle. A native DevOps-centric approach to problem resolution puts the focus on a wider range of possible error sources (including code commits), makes use of DevOps metadata to clearly define the source of the problem, and leads to a quick problem resolution. We propose such a continuous quality assurance approach, and we demonstrate it by preliminary experiments in our public Container Cloud environment and in a private OpenStack® cloud environment.

Introduction

The marriage of cloud computing and DevOps is revolutionizing the way software is delivered as a service running in a cloud environment. Cloud computing and, specifically, the infrastructure-as-a-service (IaaS) layer provide a programmable API (application programming interface) to provision compute, storage, and network components. As such, it enables the full automation of software delivery in a cloud environment.

The DevOps movement [1] proposes a method that is based on communication and collaboration between software developers and IT (information technology) personnel to improve the quality and speed of the entire software lifecycle. Automation is advocated to increase deployment frequency, promising faster time to market, lower failure rates, and shorter time to recover from failures.

New "born-on-the-cloud" programming models and architectural patterns (also known as microservices [2, 3] principles) facilitate agility by promoting fine-grained, loosely coupled services that communicate through REST (Representational State Transfer) APIs. Loose coupling enables independent evolution of each service (by autonomous development teams). It is achieved by means of dynamic service binding, dynamic service configuration, and data denormalization to avoid data dependencies. Thus, microservices can be updated independently and much more frequently and easily, as no cross-service orchestration is necessary.

Together, DevOps and the new microservice architecture style have revolutionized the way software is delivered on the cloud [4]. Companies whose software is primarily developed on the cloud for providing online services, such as Netflix** and AirBnB, have successfully applied these DevOps and microservice principles and have demonstrated the ability to deliver software daily, rather than in cycles on the order of months, which are typical of traditional IT environments [5]. As a result, these companies can perform quick market experiments to gain insight about their users and define the next set of functions to be delivered. Moreover, they can scale very quickly the number of users, not only by consuming more cloud infrastructure resources, but also by completely restructuring the software architecture, if needed, to better suit the growing demand.

Digital Object Identifier: 10.1147/JRD.2016.2517498

In this paper, we describe a set of foundational services that together form a platform that enables the delivery of software as an online service following the principles of microservices and DevOps. We further claim that to sustain the speed of software delivery enabled by the cloud, DevOps, and microservice-based design, new problem determination methods must be developed to ensure that problems on the continuously deployed services are quickly found and resolved, so that the quality of the services is not compromised. We propose such a method that 1) relies on a built-in cloud data collection mechanism (such as [6]), 2) uses DevOps pipeline metadata to clearly define the problem search space, and 3) applies correlation techniques (e.g., temporal correlation) to help quickly identify the root cause of a problem. We implemented and experimented with our method in two different environments: 1) our public Container Service (available through Bluemix* [7]) extended with foundational services for "born-on-the-cloud" software delivery [8], and 2) a private OpenStack** [9] cloud environment, where UrbanCode Deploy with Patterns (UCDwP) [10] is used to automate software delivery.

In what follows, we briefly describe DevOps and microservice principles. Then, we present our platform supporting them, summarize the types of DevOps analytics our platform enables, and delve into our problem determination approach and experiments. Before concluding the paper, we review related research efforts.

Background: The principles of "born-on-the-cloud" software delivery

In this section, we give a short overview of the principles for delivering software in a fast-paced style. The set of foundational services we describe later in the paper is based on these principles, and the problem determination techniques we present exploit the unique characteristics of the services.

Programmable infrastructure refers to a cloud environment where compute, storage, and network can be provisioned programmatically through APIs. Standard formats, such as HOT (Heat Orchestration Template) [11], can be used to specify a single request that refers to an entire topology of cloud and software resources. This principle is a basic enabler for the end-to-end automation also known as infrastructure-as-code.

Automated pipeline, also known as the "environment-as-code" principle, dictates that all code and configuration changes be automated and chained. In particular, code is automatically built and deployed upon commit, and configuration is automatically applied using domain-specific languages such as Chef [12], Puppet [13], UrbanCode Deploy [14], or others. The infrastructure is never modified manually ("no-touch infrastructure"), but only through automation code, which is tracked and versioned like the application code. Every change is reproducible and can be undone.

"*Born-on-the-cloud*" *architecture (microservices)* refers to the architectural style based on the understanding that a main concern for the developer is the maintainability and evolution of the code as a service. Loose coupling is the key that enables independent evolution of each service, which is achieved by packaging individual business functions into separate services accessed by REST APIs (each such service is termed a microservice). To obviate the need for deployment orchestration across multiple microservices, the following practices are observed: usage of a service registry to dynamically bind microservices (as opposed to static configuration files that quickly become sources of errors); usage of a configuration service to externalize important dynamic configuration that microservices may depend on; and data denormalization so that each microservice accesses only its own data, avoiding data dependencies. The full breadth of these new emerging patterns is beyond the scope of the paper (see [3, 4, 15]).

Finally, based on the principle of *rolling forward or back and never updating in place*, changes are always delivered as full-stack deployments. A new environment is created next to the current one and tested by routing a percentage of the traffic to it. The amount of traffic to the new environment is gradually increased up to 100%.

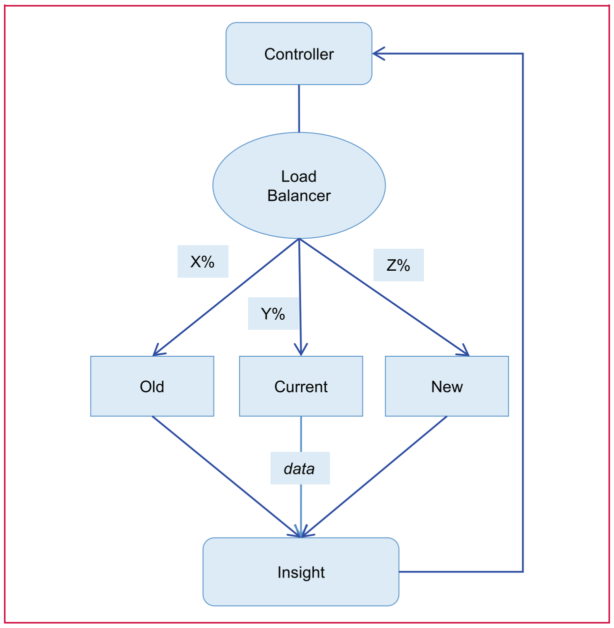

Figure 1

Three versions of a microservice, each taking a percentage of the traffic.

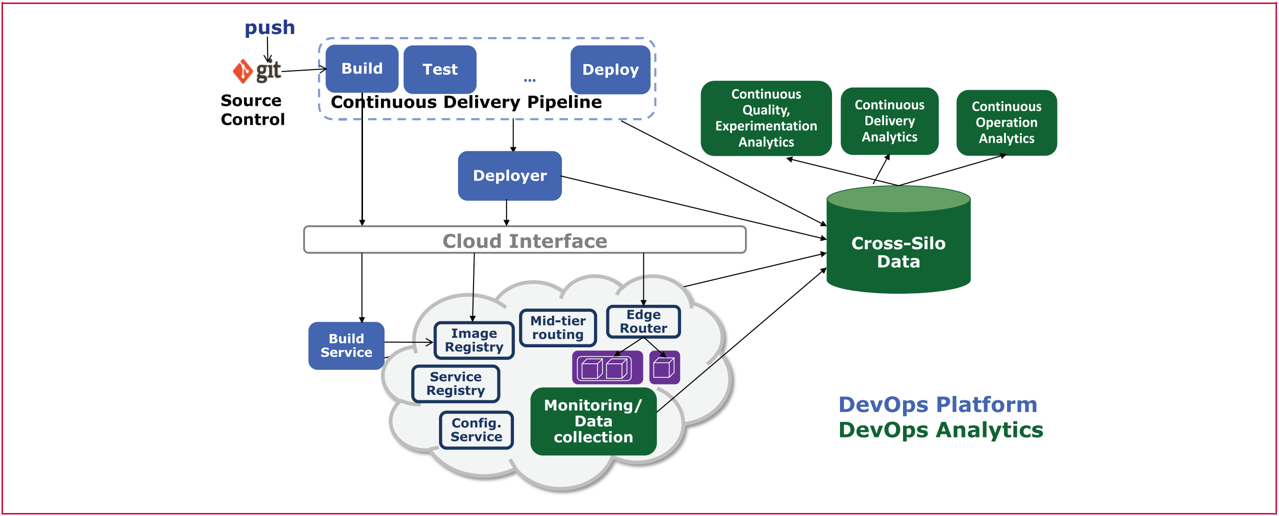

Figure 2

System architecture: DevOps and DevOps analytics platform.

The previous environment is kept in place as a backup so that, if a problem is found, it is always possible to roll back (see **Figure 1**). A foundational service that implements this principle, Active Deploy [8], was delivered by our team and is available on Bluemix [7].

It is easy to see how these principles work together to radically simplify the scaling, management of failures, and independent evolution of each individual microservice. However, we claim that, in order to fulfill the promise of extreme agility with quality, problem determination, and incident management must be approached differently. Traditional IT service management techniques assume a relatively static environment and, in many cases, are compartmentalized: operations data is not correlated with code development data. Moreover, even in the operations space, data collection mechanisms frequently are specific to individual runtime environments, resulting in fragmented views and high setup overhead. Adopting these prior techniques in fast-paced environments based on the above principles would be counterproductive, creating an impediment to agility. We therefore propose an approach for problem determination that takes advantage of the specific characteristics of fast-paced environments. Our approach relies on a built-in cloud data collection platform for continuously collecting logs, runtime data, metrics, and other types of data from all components of distributed applications and across all layers of the software stack. Such a data collection mechanism is built into the Bluemix platform [6]. Our approach uses context and metadata available from an automated pipeline that delivers the software, enabling a significant reduction of the problem search space. In the next sections, we will describe the platform we built and the experiments we have performed with this new approach to detection of problems in fast-paced cloud software-delivery environments.

Cloud DevOps and DevOps analytics platform

In this section, we describe the essential building blocks to enable both agility and analytics-driven quality in delivering cloud applications. As shown in **Figure 2**, at the center of our system is the target cloud environment, which hosts both user applications and foundational services used by applications developed using microservice principles. This environment consists of several layers from the hardware resources, the systems software that constitutes the infrastructure-as-a-service cloud computing platform, and, in some cases, an additional platform-as-a-service layer. In our experiments, we target two cloud environments: IBM's Bluemix public-cloud environment, offering Docker containers as infrastructure (Container Cloud) and microservice-centered DevOps tools, and a private cloud environment that uses OpenStack-based virtual machines and more traditional enterprise DevOps tools. Our approach is equally applicable to both.

The DevOps platform consists of tools that provide the abstractions, control, and automation necessary to continuously build, test, deploy, manage, and improve cloud applications. Automation, integration, traceability, and ease-of-use are the key characteristics of this platform, enabling agility. The continuous delivery pipeline shown in Figure 2 provides users a mechanism to create a customized, automated, and repeatable flow to introduce code changes into production, beginning with code commits made to a Git revision control repository. Predefined stages in this pipeline can include different types of testing for security and quality, software artifact and image build, deployment, and live testing. Restricting changes to be introduced only through such a pipeline ensures consistency and traceability in addition to the speed enabled by automation. IBM provides a delivery pipeline service targeted at individual microservices for its Bluemix cloud environment [16]. Other examples include Jenkins [17], a very popular open-source self-serve solution, and hosted services like TravisCI [18] and CloudBees** [19].

The delivery pipeline we adopt for our experiments with the public Container Cloud uses two other building blocks that constitute our DevOps platform: a Build Service and the Deployer (the Active Deploy Service [8]). The Build Service creates and pushes Docker images to the registry used by our target cloud environment. The Deployer enables users to update their running applications without incurring downtime. It achieves this goal by providing policies to introduce new versions and manage request traffic between them so that easy roll-forward or roll-back are possible depending on whether the new deployment was satisfactory or not. Like the delivery pipeline, the Deployer is also currently targeted at microservices that are stateless and can be individually managed. For deployments that require complex orchestration between multiple components of an application, tools like UCDwP [10] are commonly used. We present scenarios using both UCDwP and Active Deploy.

Active Deploy supports several automated policies for updating microservices such as timed, blue-green deploy [20]. Netflix's Asgard [21], CodeDeploy from Amazon Web Services** [22], and Application Upgrade for Microsoft Azure** [23] address the same needs as Active Deploy, varying in the scope and features provided.

Aside from our DevOps platform, Figure 2 also refers to our DevOps Analytics platform, which is responsible for collecting operational data from the cloud environment, including logs, metrics, and runtime data, as well as data and metadata across the entire lifecycle of cloud applications from the continuous delivery pipeline and the Deployer. All collected data is indexed and made searchable in a cross-silo data repository on top of which analytics solutions, including those for problem determination, can be built. In our system, we use Elasticsearch [24] as our cross-silo data repository. Collecting data from the underlying cloud (Container Cloud or OpenStack) and from users' containers and VMs (virtual machines) is accomplished by data crawlers that run as part of our infrastructure, invisible to users. In addition to Elasticsearch, other open-source components we use in our data collection and DevOps analytics substrate are Apache Kafka [25], Logstash** [26], and Kibana** [27]. In the next section, we discuss different types of analytics at different points in the DevOps lifecycle with a more in-depth exploration of one of them. Also, we provide more details on our DevOps analytics and data collection architecture when describing our experiments.

DevOps analytics

In the new world of fast-paced IT, there are two main challenges: making sure agility does not compromise quality and that slow and tedious processes to address operational issues do not hamper agility and uninterrupted operation. Assimilating, correlating, and analyzing data across the entire lifecycle to generate actionable insights is the key to sustaining agility and achieving operational excellence. Next, we briefly describe three broad categories of DevOps analytics, shown as three green boxes at the top right of Figure 2.

Continuous delivery analytics aim to quantify and improve the overall efficacy of the entire DevOps Process. The main goal is to increase the frequency of code change and reduce mean time to delivery. Tools and techniques aimed at tracking and reporting, finding and improving bottlenecks, and predictive modelling to enable what-if explorations fall into this category.

Continuous quality and experimentation refers to the analytics and tools necessary to automate, speed up, and enable quality control processes like pre-deployment and post-deployment testing for correctness, security compliance, performance, adherence to service-level agreements, and other relevant metrics. Traditionally, these processes take hours, if not days, and can involve significant human effort. Users are also increasingly interested in immediate feedback on new experimental features as the deployed code evolves, which requires correlation and analysis of code change, deployment, as well as operational data.

Continuous operation refers to the analytics and tools aimed at ensuring continuous operation by minimizing downtime and the impact of outages. Tools and techniques for problem detection, determination, resolution, and avoidance fall into this category. Data silos and inherent complexity are the main hindrances to agility in this area. An important side-effect of data silos is the limited resources available for real-time problem determination without having to cross department or organizational boundaries. Our hypothesis is that the complexity in automating problem determination, resolution, and avoidance can be reduced by using DevOps data and metadata (deployment context). Next, we explore this hypothesis via experiments to perform problem determination for full-stack deployments in the context

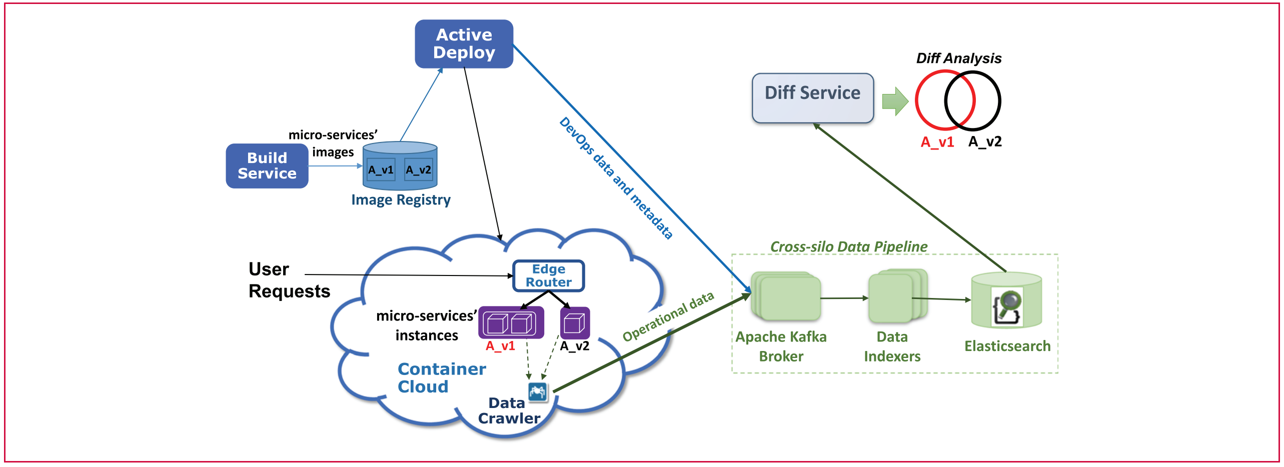

Figure 3

Problem determination with Active Deploy and the diff service (A_v1: version 1 of microservice "A"; A_v2: version 2 of microservice "A").

of two different cloud environments, using different sets of DevOps tools. We especially focus on empowering developers rather than IT operators.

Problem determination for Active Deploy and IBM Containers

We now discuss our early prototype for quickly diagnosing problems that may happen when new versions of microservices are being built, deployed, and gradually subjected to request traffic. Our prototype, illustrated in **Figure 3**, targets the IBM Container Cloud and uses Active Deploy to transition from the current version of a microservice to a newly introduced one without taking the service offline. The Docker images for the microservice are built by our Build Service. Active Deploy serves the Deployer function explained before and depicted in Figure 2.

A key component of Container Cloud is a data crawler (see Figure 3), which observes all running containers from their hosts and periodically collects containers' log events from log files identified as relevant, usage metrics, and runtime data on the file system in general (and configuration files in particular), operating system packages, running processes, and network connections. All data is sent to a data pipeline whose entry point is an Apache Kafka cluster. Further down the pipeline, the data is fetched from Kafka and properly indexed onto Elasticsearch, where it becomes searchable.

Data types, metadata, and features

Critically, each piece of collected data is properly typed. For instance, our types of operational data include "file" (representing file system objects), "config" (representing files identified as configuration files), "process" (corresponding to operating system processes), "package" (for operating system packages), "os" (representing general operating system information), and "connection" (encapsulating information on network connections). Each data type has a set of known attributes whose values define an instance thereof. We refer to each data type instance as a "feature."

Besides its data attributes, each feature contains important metadata. In particular, each feature is associated with a "namespace" attribute, which is a logical representation of the container from where the feature originated. One example of namespace would be *microserviceA/v1/dbff7c4181d9*. In this case, the namespace identifies the name of the microservice, its version, and the ID of the container in question. Aside from namespace, other important metadata associated with features include the ID of the Docker image from which the container was created, the image version, the name and ID of the user owning the container, and a timestamp corresponding to the time when the features were collected by the data crawler.

As one can realize, some metadata attributes associated with a feature annotate it with DevOps-related information, such as microservice names and versions, and Docker images and versions, augmenting the data with a deployment context. In addition, we also define DevOps data types, such as "dockerfile," used to index the information pertaining to Dockerfiles that prescribe how to build images.

The diff service

The diff service, also shown in Figure 3, provides users with a high-level abstraction layer for querying and analyzing all cross-silo indexed data. In a nutshell, it supports two main functions: retrieving the state of any container at any point in time, and performing analytics to compute state differences (“diffs”).

After being indexed on Elasticsearch, a feature (data type instance) can be queried by the values of any of its data and metadata attributes. As namespaces identify microservices, versions, and containers in the entire Container Cloud, queries on namespaces can be thought of as spatial queries. Temporal queries and reasoning, on the other hand, are enabled in two different ways. First, the timestamp attribute associated with each feature can be used to retrieve the entire state of a container at any given point in time. Second, the diff service allows tagging DevOps-related, significant points in time. Thus, aside from using timestamps to perform queries, temporal queries can also be performed by using these user-created, semantically significant “bookmarks.” The diff service keeps a mapping between each bookmark and the timestamp corresponding to when it was created, allowing the users to focus on relevant events rather than raw timestamps. Examples of bookmarks created by the DevOps pipeline include those indicating when Docker images have been built, and when a microservice version has been deployed.

In this context, we define the state of a container as a snapshot containing a set of features originating from that container tagged with exactly the same timestamp assigned by the data crawler. Clearly, the frequency at which the crawler collects data determines the amount of snapshots available and hence the supported time granularity for state queries. When asked for a container’s state given a timestamp as input, the diff service maps the input timestamp to the closest discrete snapshot the crawler has collected based on the indexed timestamps. The user may indicate to the diff service, as input, whether to return the closest snapshot considering both directions (past and future), or one direction only (past or future).

Individual containers can be reasoned about in terms of their corresponding namespaces, which typically link them to specific microservices and versions. From this perspective, keeping in mind the aforementioned discretization of the state, the diff service can determine, for example 1) the state of a namespace, considering all feature types or a subset, at any point in time, (2) for a given namespace, how its state has changed (state diff) within a time interval, considering all feature types or a subset, and 3) given two pairs of the type (namespace, timestamp), the differences between the namespaces’ states taken at the respective points in time, considering all feature types or a subset. Analogous queries can be performed using bookmarks rather than timestamps.

When computing the state diff for two distinct namespaces or for the same namespace within a time interval, the diff service returns three sets of data: (1) features present in the first state snapshot, but not in the second; (2) features present in the second state snapshot but not in the first; and, (3) features present in both state snapshots, but containing different values for at least one attribute, and what the different values are.

The functionality provided by the diff service has many potential applications. When two versions of a microservice are running side-by-side, it would be appealing to compare them, especially when a problem arises with the new version and a roll-back is required. Next, we report on an experiment of this type we have performed, exercising all building blocks depicted in Figure 3.

Using the diff service: Performance problem in new microservice version

In this experiment, we used a Java** application written according to the microservice architecture principles, along with a Dockerfile prescribing how to build a Docker image for our microservice so that we could deploy it on Container Cloud. The next paragraphs describe the scenario portrayed by our experiment.

The Build Service produces the version “v1” of the image of microservice “A,” labeled “A_v1” in the image registry depicted in Figure 3. Using Active Deploy, developers deliver the version “v1” of microservice “A,” labeled “A_v1,” running as containers on Container Cloud.

Sometime later, a defect is filed with respect to the microservice. The developers deem this defect as a low-risk, easy fix. The defect owner fixes the defect and, when satisfied with her local tests, triggers a new build. Rigorous tests are skipped because of the low risk and difficulty attributed to the defect. After the build produces image “A_v2,” the developer instructs Active Deploy to deploy it and gradually shift traffic from “A_v1” to “A_v2.” Confident that the easy-fix is done and no problems will occur, the defect owner leaves office while Active Deploy is managing the transition from “v1” to “v2.”

Several hours later, the developer on duty on the night shift starts being notified of performance-related exceptions on “A_v2,” which is now taking some live user traffic. The developer in charge uses Active Deploy to roll-back to A_v1, so that A_v2 is no longer taking any live traffic, and then proceeds to diagnose the problem with A_v2. Looking for clues, the developer in charge resorts to the diff service to learn the differences between containers “A_v2” and “A_v1.” He asks the diff service to diff (“A_v2,” “A_v2_deployment”) and (“A_v_1,” “A_v1_deployment”), where the second element of each pair is the bookmark name associated with the end of deployment of each respective microservice version.

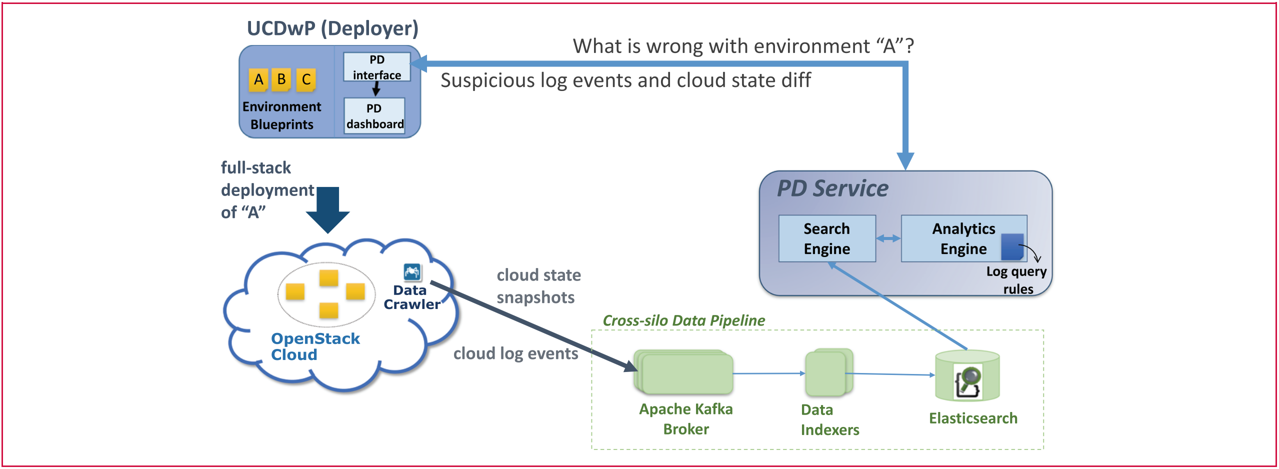

Figure 4

Problem determination with UCDwP and the PD (problem determination) service.

The diff service analytics indicates that a Java process running on container "A_v1" was not on "A_v2," and a Java process running on "A_v2" was not on "A_v1." Inspecting the command-line attribute of each process feature revealed that the only difference between them was the addition of "-Xmx250m" to "A_v2"'s process command line, indicating that the heap size of "A_v2" was limited to 250 MB, which was far less than the default value of 1 GB used in "A_v1." Furthermore, the other difference found by the diff service was that the "A_v2" container had a file on its file system that was not present on "A_v1," namely, "/root/wlp/usr/servers/daytrader/jvm.options." This indicates that the developer who made the change had in his work environment a "jvm.options" file for testing purposes. This file contains values for options used to configure a JVM** (Java Virtual Machine**). The file was not related to the defect fix introduced in A_v2, and had been inadvertently committed and picked up by the Java process of the new microservice version, leading to the "-Xmx250m" flag in the process command line.

Problem determination for UCDwP and the OpenStack cloud

Next, we discuss our extension of an IBM product in the DevOps arena, enabling it to help users diagnose problems that occur at full-stack deployment time in private OpenStack clouds (see **Figure 4**). This product, IBM UCDwP, automates the deployment of entire software stacks in an OpenStack private cloud setup. It serves the Deployer function described earlier in the DevOps platform and consumes deployable application components produced by the build stage of the delivery pipeline. Each application artifact is associated with automation code, typically written using an infrastructure-as-code tool (e.g., Chef [12]), that encapsulates the logic to deploy and configure the corresponding application component. UCDwP environment blueprints (illustrated as "A," "B," and "C" in Figure 4), which are written in the HOT [11] domain-specific language, reference application artifacts and deployment/configuration automation code. When the deployment of an environment is triggered, the user is asked to provide values for several deployment parameters exposed by the environment blueprint, which typically include choosing the target cloud, flavors of cloud resources, and specific versions of the application artifacts. Finally, UCDwP orchestrates the full-stack deployment, by interacting with the target cloud to provision the requested cloud resources, and interacting with the automation tool to execute the various pieces of automation logic responsible for deploying and configuring the application components.

When deploying an entire application environment, many problems may arise. Problems may occur, for instance, due to bugs in the automation logic (e.g., Chef recipes), invalid usage of cloud resources, and cloud infrastructure capacity, misconfiguration, or transient errors. Troubleshooting requires a holistic knowledge of the entire stack and infrastructure, including the build system, the application code, and the deployment automation logic, as well as all cloud subsystems, which can overwhelm and frustrate the team. To mitigate the troubleshooting complexity in such an intricate setting, it is imperative that insights for problem determination be given to users.

To that end, we implemented an extension to the UCDwP DevOps pipeline aiming to reduce the time to diagnose problems. We have added a problem determination (PD) service to the pipeline and changed the UCDwP product so as to make it directly interact with the PD service to get insights into problems pertaining to the context of specific application environments. Our proposed PD service currently supports OpenStack as the target cloud.

Problem determination service

Figure 4 illustrates the interplay between the PD service and UCDwP, as well as our data crawler embedded into a private OpenStack cloud. The data crawler collects log events produced by the various cloud subsystems (compute, storage, network, etc.). The crawler is configured to properly parse the log events from each monitored OpenStack subsystem, annotating each event with semantic information extracted from it, such as the log level, the component from where the event originated, and codes returned by corresponding OpenStack calls. Moreover, all information pertaining to an exception is coalesced and indexed as a single event properly tagged, and each event is associated with the originating target OpenStack cloud. These annotations allow the PD service to later retrieve log events by running queries that carry relevant semantic meaning. For example, the service would be able to retrieve all log events either tagged as exceptions or identified as a log level of *error* from a specific OpenStack subsystem belonging to a specific OpenStack target cloud.

Aside from collecting OpenStack log events, the data crawler also takes snapshots of the OpenStack cloud state, capturing the current state of storage volumes, networking and pools of IP addresses, virtual images, security groups and rules, and tenants.

As shown in Figure 4, the main components of the PD service are the search engine and the analytics engine. The former implements the logic to access and query all indexed data (OpenStack log events and state snapshots). The latter interacts with the search engine to query the data and extract from it the pieces relevant to the problem being diagnosed. The analytics engine is engaged, for instance, when the PD service is answering a REST call to retrieve suspicious log events and cloud state changes (“diff”) that may be associated with a problem that caused a deployment failure in a particular application environment. As a result, the engine not only selects relevant, suspicious log events, but it also returns, when applicable, cloud state changes derived from a subset of the “diff” between cloud state snapshots. When selecting relevant, suspicious log events, the analytics engine contextualizes queries (e.g., by scoping the query with a time interval that may be related to the problem being diagnosed), and correlates the query results with information in the environment blueprint (e.g., referenced virtual images, security groups, etc.). Queries can be provided by the PD service users or generated by the analytics engine.

When showing the list of deployments, UCDwP presents to the user the option of “diagnosing” the problem of a failed deployment by selecting an icon. That action triggers a REST call to the PD service, requesting suspicious cloud log events and suspicious cloud state changes relevant to the context of the used blueprint and the deployment start, end, and last-update times. The PD service can return as a result either the raw selected log events and cloud state changes or a link to a customized Kibana [27] dashboard. The Kibana dashboard is dynamically created by the service to render the data showing the time correlation of a few selected log events from different cloud subsystems, as well as a few selected changes in cloud state when applicable. The dynamically-created Kibana dashboard customized for the deployment problem in question is depicted by the rectangle labeled “PD dashboard” in Figure 4.

Using the problem determination service: Storage capacity problem

In this scenario, using UCDwP, we created an environment blueprint specifying the installation of a Java application onto a VM to be provisioned from a Fedora 20 virtual image and connected to a public network. The blueprint also indicated that a storage volume of 12 GB must be created and attached to the VM. Before starting the deployment of an actual environment from this blueprint, we selected the OpenStack “m1.small” flavor for the VM. Next, UCDwP triggered the deployment, interacting with OpenStack’s orchestration engine to provision the required resources, which was interrupted with a generic error message.

Following what users would normally do, we looked for clues on the OpenStack dashboard, which showed that the stack we had tried to deploy was in failed state. Looking for further details, we noticed that the VM was successfully created but the storage volume could not be created by OpenStack. The error message next to the volume stated the following: “ResourceInError: Went to status error due to Unknown.”

We then invoked the PD service through UCDwP to obtain more insights into the problem. As a result, a dynamically-created Kibana dashboard appeared on the UCDwP GUI. The PD service selected logs from only two OpenStack subsystems, namely, Heat (orchestration engine) and Cinder (block storage service). The first Cinder log event selected (which appears at the top of the list) by the PD service read “Insufficient free space for volume creation... (requested/avail): 12/7.0.” This log message implies that although we have requested a 12-GB storage volume, the server had only 7 GB available and thus could not fulfill the request. That was indeed the root cause of the problem.

The dashboard for diagnosing this problem also exhibited on a timeline clusters of three log messages, one from Heat and two from Cinder. Each cluster was about 20 seconds apart, representing multiple failed attempts at creating a Cinder storage volume initiated by Heat. The selected Heat log event was an exception when trying to create the volume, whereas the three Cinder log messages included the aforementioned one at the top of the list.

Using the problem determination service: Security group problem

This scenario also starts with a blueprint for deploying the previously used application onto a VM. This time, however, the blueprint specifies that the VM be added to a security group whose rule dictates that 8080 is the only network port that must be open, which is the one exposed by the application for users' network connections. Deploying the application environment through UCDwP succeeds, and the application works as expected.

Sometime later, a new version of the application-deployable artifacts is produced by the build system. Before we started deploying a new application environment with the latest version, an OpenStack administrator inadvertently edited the security group referenced by the environment blueprint, commanding that the port 8080 be closed. Although the deployment of the application succeeds, users can no longer establish connections with the application web server. Given that there were no OpenStack errors, we immediately called the PD service from UCDwP to diagnose that particular environment deployment. The resulting Kibana dashboard selected one log event from Nova, the OpenStack compute service, which read "Revoke security group ingress MTM_security," where MTM_security was the name of the security group referenced by the environment blueprint. Furthermore, this time the Kibana dashboard dynamically generated by the PD service also showed OpenStack cloud state changes indicating that the security rule with the following attributes "from_port=8080, to_port=8080, ip_protocol=tcp, ip_range=0.0.0.0/0" had been deleted from the security group "MTM_security." Both findings were consistent with the root cause of the problem.

In both scenarios described above, without the PD service, users would have to start by inspecting the OpenStack dashboard. Finally, they would have to identify relevant log files out of dozens, probably spread across several OpenStack hosts, and examine many log events, all assuming there is sufficient OpenStack expertise and knowledge of the environment blueprint.

Related work

Although little work so far has studied how to make effective use of DevOps data combined with operational data for problem determination, many research efforts have been made to address problem determination in different contexts. Huang et al. [28] proposed a tool that attempts to automate problem determination based on rules provided by human experts. Other research efforts apply causality analysis to help identify misconfigurations or the root cause of performance anomalies [29–31]. Alternatively, to address misconfiguration problems in Windows Registry, PeerPressure [32] compares a problematic configuration state with a large set of healthy configurations, whereas Lao et al. [33] combine high-level symptom descriptions and low-level system state.

Some studies emphasize data collection for problem determination. Cohen et al. [34] devised a method for indexing, clustering, and retrieving a system's state in order to distill the signature of observed problems so that a recurring problem can be detected by similarity. Verbowski et al. [35] proposed a method for monitoring, efficiently storing, and analyzing all system's persistent-state interactions to improve systems management, including problem diagnosis. These efforts do not consider data from DevOps processes.

Representative of a body of research on tracing requests as they flow through the system for failure detection, Pinpoint [36] applies clustering techniques on the traced requests to determine what components are likely to be the root cause of failures.

Similar in spirit to testing multiple versions of a microservice in production to decide whether or not an update is satisfactory, a technique for validating human operator actions has been proposed [37, 38] where the system components operated on are tested with live requests and compared with functionally equivalent components that are known to be working properly.

Many research efforts have been made to use logs as a means to detect problems. For instance, Xu et al. [39] propose a technique to parse log events based on source-code analysis, and convert the parsed events into machine learning features that are then subjected to a machine learning algorithm for detecting operational problems. On the other hand, Distalyzer [40] uses machine learning to compare two sets of logs, one for a normal system and one for a system exhibiting performance problems, to determine associations between systems components and performance. Our PD service, on the other hand, correlates log events with DevOps data and with cloud state in order to pinpoint suspicious events.

In a different vein, Yuan et al. [41] have proposed a misconfiguration troubleshooting technique that identifies invariant configuration access rules and predicts what configuration access to expect given a stream of accesses. Considering the context of the entire software stack and its operating environment, EnCore [42] exploits the interaction between configurations and the executing environment as well as correlations between configurations in order to learn configuration rules from a set of configurations. EnCore uses an approach similar to ours for collecting operational data.

A body of research has been conducted in the software evolution domain to analyze code changes as new code is committed to revision control repositories. Some research in this domain combine release history data and code commits to help understand how the code and its intra-dependencies change over time [43, 44]. On the other hand, our work combines information from traditionally isolated sources, namely DevOps and operations, to help users understand problems.

Finally, Weaver [45], a DevOps domain-specific language and runtime, was designed to orchestrate the execution of deployment automation and the provisioning of cloud resources. Weaver allows the definition of pre-deployment validation rules that check for potential problems before they happen. Although pre-deployment validation is helpful, its scope is limited by the rules that someone needs to manually write. Our approach of highlighting changes in the operational environment contextualized by DevOps metadata does not rely on manually written rules.

Conclusion

We have proposed a set of cloud foundational services that facilitate the extreme agility needed for delivering software as a service in fast-paced environments according to the principles of DevOps and microservices. To truly fulfill the DevOps promise of agility, even in light of problems and failures, we have also proposed an approach to problem determination, native to such fast-paced environments, that takes advantage of the DevOps processes to contextualize virtually all data produced by a cloud data collection substrate. With such a contextualization, our problem determination services can reduce the problem search space and correlate traditional operational data with DevOps data and metadata, presenting insights to the user to allow quicker problem resolution.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Netflix, Inc., OpenStack LLC, CloudBees, Inc., Amazon Technologies, Microsoft Corporation, Elasticsearch Corporation, or Oracle America in the United States, other countries, or both.

References

1. M. Huttermann, *DevOps for Developers*. New York, NY, USA: Apress, 2012.
2. M. Fowler, Microservices. [Online]. http://martinfowler.com/articles/microservices.html.
3. S. Newman, *Building Microservices*. North Sebastopol, CA, USA: O'Reilly, 2015.
4. A. Schaefer, M. Reichenbach, and D. Fey, "Continuous integration and automation for DevOps," *IAENG Trans. Eng. Technol.*, vol. 170, pp. 345–358, 2013.
5. "The Netflix Tech Blog," Netflix, Los Gatos, CA, USA. [Online]. Available: http://techblog.netflix.com/2012/12/videos-of-netflix-talks-at-aws-reinvent.html.
6. Agentless System Crawler. [Online]. Available: https://developer.ibm.com/open/agentless-system-crawler/.
7. Bluemix. [Online]. Available: http://www.ibm.com/cloud-computing/bluemix/.
8. The IBM Active Deploy Service. [Online]. Available: https://www.ng.bluemix.net/docs/services/ActiveDeploy/index.html.
9. OpenStack. [Online]. Available: https://www.openstack.org/.
10. IBM UrbanCode Deploy with Patterns. [Online]. Available: https://developer.ibm.com/urbancode/products/urbancode-deploy-with-patterns/.
11. Heat Orchestration Template (HOT). [Online]. Available: http://docs.openstack.org/developer/heat/template_guide/hot_spec.html.
12. Chef. [Online]. Available: https://www.chef.io/.
13. Puppet. [Online]. Available: https://puppetlabs.com/.
14. IBM UrbanCode Deploy. [Online]. Available: http://www-03.ibm.com/software/products/en/ucdep.
15. M. T. Nygard, *Release it!: Design and Deploy Production-ready Software*. Frisco, TX, USA: Pragmatic Bookshelf, 2007.
16. IBM Bluemix Delivery Pipeline Service. [Online]. Available: http://www.ibm.com/developerworks/topics/delivery__pipeline__service/index.html.
17. Jenkins. [Online]. Available: https://jenkins-ci.org/.
18. Travis CI. [Online]. Available: https://travis-ci.org/.
19. CloudBees. [Online]. Available: https://www.cloudbees.com/.
20. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Reading, MA, USA: Addison-Wesley, 2010.
21. Netflix Asgard. [Online]. Available: https://github.com/Netflix/asgard.
22. Amazon Web Services CodeDeploy. [Online]. Available: http://aws.amazon.com/codedeploy/.
23. Microsoft Azure. [Online]. Available: http://azure.microsoft.com/.
24. Elasticsearch. [Online]. Available: https://www.elastic.co/products/elasticsearch.
25. Apache Kafka. [Online]. Available: http://kafka.apache.org/.
26. Logstash. [Online]. Available: https://www.elastic.co/products/logstash.
27. Kibana. [Online]. Available: https://www.elastic.co/products/kibana.
28. H. Huang, R. Jennings III, Y. Ruan, R. Sahoo, S. Sahu, and A. Shaikh, "PDA: A Tool for Automated Problem Determination," in *Proc. LISA Conf.*, Dallas, TX, USA, 2007, pp. 153–166.
29. Y. Su, M. Attariyan, and J. Flinn, "AutoBash: Improving configuration management with operating system causality analysis," in *Proc. ACM SOSP*, Stevenson, WA, USA, 2007, pp. 237–250.
30. M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proc. Symp. OSDI*, Vancouver, BC, Canada, 2010, pp. 1–14.
31. M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *Proc. Symp. OSDI*, Hollywood, CA, USA, 2012, pp. 307–320.

32. H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with Peerpressure," in *Proc. Symp. OSDI*, San Francisco, CA, USA, 2004, pp. 245–257.
33. N. Lao, J.-R. Wen, W.-Y. Ma, and Y.-M. Wang, "Combining high level symptom descriptions and low level state information for configuration fault diagnosis," in *Proc. LISA Conf.*, Atlanta, GA, USA, 2004, pp. 151–158.
34. I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proc. ACM SOSP*, Brighton, U.K., 2005, pp. 105–118.
35. C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev, "Flight data recorder: Monitoring persistent-state interactions to improve systems management," in *Proc. Symp. OSDI*, Seattle, WA, USA, 2006, pp. 117–130.
36. M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *Proc. Symp. NSDI*, San Francisco, CA, USA, 2004, pp. 1–14.
37. K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *Proc. Symp. OSDI*, San Francisco, CA, USA, 2004, pp. 61–76.
38. F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and Validating Database System Administration," in *Proc. Usenix Annu. Tech. Conf.*, Boston, MA, USA, 2006, pp. 213–228.
39. W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. ACM SOSP*, Big Sky, MT, USA, 2009, pp. 117–132.
40. K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. Symp. NSDI*, San Jose, CA, USA, 2012, pp. 1–14.
41. D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, "Context-based online configuration-error detection," in *Proc. Usenix Annu. Tech. Conf.*, Portland, OR, USA, 2011, pp. 1–14.
42. J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting system environment and correlation information for misconfiguration," in *Proc. Conf. ASPLOS*, Salt Lake City, UT, USA, 2014, pp. 687–700.
43. M. Fisher and H. Gall, "EvoGraph: A lightweight approach to evolutionary and structural analysis of large software systems," in *Proc. IEEE WCRE*, Benevento, Italy, 2006, pp. 179–188.
44. M. Pinzger, M. Fischer, and H. Gall, "Towards an Integrated view on architecture and its evolution," in *Electron. Notes Theoretical Comput. Sci.*, vol. 127, no. 3, pp. 183–196, Apr. 2005.
45. M. Kalantar, F. Rosenberg, J. Doran, T. Eilam, M. Elder, F. Oliveira, E. Snible, and T. Roth, "Weaver: Language and runtime for software defined environments," *IBM J. Res. Dev.*, vol. 58, no. 2/3, pp. 10:1–10:12, 2014.

Received June 4, 2015; accepted for publication June 29, 2015

Fábio Oliveira *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (fabolive@us.ibm.com)*. Dr. Oliveira is a Research Staff Member at the IBM T. J. Watson Research Center. He earned a Ph.D. degree in computer science from Rutgers University in 2010. His research interests include systems management, distributed systems, cloud computing, and operational and DevOps analytics.

Tamar Eilam *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (eilam@us.ibm.com)*. Dr. Eilam is an IBM Fellow working on next generation cloud, DevOps, and DevOps analytics. She received her Ph.D. degree in computer science from The Technion, Israel Institute of Technology in 2000.

Priya Nagpurkar *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (pnagpurkar@us.ibm.com)*. Dr. Nagpurkar is a Research Staff Member and Manager of the Cloud DevOps group at the IBM T. J. Watson Research Center. She received her Ph.D. degree in computer science from the University of California, Santa Barbara, in September 2007. Her research interests include program analysis, debugging, problem determination, performance analysis and optimization, and DevOps analytics.

Canturk Isci *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (canturk@us.ibm.com)*. Dr. Isci is a Research Staff Member in the Cloud Computing division at the IBM T. J. Watson Research Center, where he leads the Scalable Data Center Analytics team in the Research and the Operational Analytics Squad within IBM Cloud Services. He received a B.S. degree in electrical engineering from Bilkent University, an M.Sc. degree with distinction in VLSI (very-large-scale integration) System Design from University of Westminster, and a Ph.D. degree in computer engineering from Princeton University. His research interests are cloud computing, operational and DevOps analytics, novel monitoring techniques based on virtualization and containerization abstractions, and energy-efficient computing at multiple levels of compute hierarchy, from microarchitectures to data centers.

Michael Kalantar *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (kalantar@us.ibm.com)*. Dr. Kalantar graduated from Cornell University and has taught at Shandong and Shiyou Universities, and he now works at IBM Research. His research interests are system management and distributed systems.

Wolfgang Segmuller *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (werewolf@us.ibm.com)*. Mr. Segmuller is a Senior Software Engineer at the IBM T. J. Watson Research Center. He has researched systems management, network management, and distributed systems for more than 30 years at IBM.

Edward Snible *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (snible@us.ibm.com)*. Mr. Snible is a Software Engineer and member of the Cloud DevOps group. His research interests include visualization of software deployments and the detection of errors in distributed systems.