Enabling DevOps for Containerized Data-Intensive Applications: An Exploratory Study

Marios Fokaefs
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, ON, Canada
fokaefs@yorku.ca

Cornel Barna
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, ON, Canada
cornel@cse.yorku.ca

Rodrigo Veleda
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, ON, Canada
rveleda@yorku.ca

Marin Litoiu
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, ON, Canada
mlitoiu@cse.yorku.ca

Joe Wigglesworth
IBM Toronto Lab
IBM Canada Ltd.
Markham, ON, Canada
wiggles@ca.ibm.com

Radu Mateescu
IBM Toronto Lab
IBM Canada Ltd.
Markham, ON, Canada
mateescu@ca.ibm.com

ABSTRACT

In an ever-changing landscape of software technology, new development paradigms, novel infrastructure technologies and emerging application domains reveal exciting opportunities, but also unprecedented challenges for developers, practitioners and software engineers. Amongst this innovation, containers as infrastructure support, data-intensive application as a domain and DevOps as a development paradigm have gained significant popularity recently. In this work, we focus on these concepts and present an exploratory study on how to develop such applications, deploy and deliver them in Docker containers and eventually manage them by enabling autoscaling on the container level. In the paper, we detail our experimental process pointing out the problems we encountered along with the solutions we used. Eventually, we present a set of stable experiments to demonstrate the autoscaling capabilities we achieved.

Keywords

devops, containers, big data, data analytics, cloud computing, adaptive systems, autoscaling

1. INTRODUCTION

With a multitude of sources to produce data in everyday life including mobile devices, sensors, smart appliances, smart cars and so on, the requirements for storing and analyzing this data to acquire useful information and knowledge have increased to exceptional levels and novel technologies are constantly being proposed to better handle the need. Innovations have appeared both on the software and the infrastructure layer and the two layers seem to simultaneously push each other forward. On the hardware side, in the beginning, we had mainframes and physical servers and deployed software was available in a client-server architecture directly to end-users. Later, software was being developed in a modular manner and functionality became available in self-contained independent pieces of software, what became known as web services. However, since web services were meant to be invoked by other software, it became evident that traffic could be quite volatile and fluid requiring additional flexibility from the infrastructure's perspective. This gave rise to the virtualization of resources, the central concept of cloud computing, so that infrastructure is easily customizable and adaptable as the requirements change. More recently, infrastructure has been raised to new levels of abstraction; containers package web services into isolated software environments (including libraries, configurations, file systems etc.) that are hosted by an operating system inside a VM. Containers constitute a more lightweight hosting environment that lifts from the software developer the responsibility of managing and maintaining the infrastructure or even the operating system, bringing development closer to the DevOps model [10].

In light of these advancements and in combination with the unique challenges that big data applications pose, the provision, deployment, management and maintenance of such software on the cloud becomes non-trivial requiring novel and effective methods. Our research focuses mainly on self-adaptive systems through scaling of virtual resources. In this particular paper, we study the challenges, pitfalls and best practices in developing and adapting data-intensive applications on extra layers of virtualization, namely Docker containers. We adopt an exploratory method, starting with vanilla configurations and addressing the problems as they come. First, we start with setting up the infrastructure and developing our testbed applications. Second, we deploy the applications in the infrastructure and stress them with intensive workload. The reason behind stressing is to create causes for scaling and identify the respective limits of the applications and the resources. Finally, based on these last observations, we set up an autonomic management system to provide autoscaling capabilities to our topologies. We have focused on two applications; a three-tier one with a simple

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

RIGHTSLINK

service that accesses a relational database and a four-tier one with a service cluster that submits analytics requests to another cluster, which in turn accesses a NoSQL database cluster.

In the process of our exploratory study, we identify the *challenges* around the DevOps phases we are interested in (development, deployment and management), we point out the *options* we explored to address each challenge and finally we present a set of experiments we were able to conduct based on our experience. Eventually, the goal of this paper is to create a DevOps guide and experience report on how to develop data-intensive applications intended for lightweight and distributed cloud systems, how to deploy them on such system and, finally, how to manage them and maintain their performance.

The rest of this paper is organized as follows. Section 2 provides an overview of the technologies and the concepts we employ, namely Docker Containers, Spark analytics, Cassandra data storage and we outline the architecture and the implementation details of the testbed applications. Section 3 discusses how we set up the infrastructure in preparation for our experiments. Section 4 presents the backbone of our exploratory process where we record the challenges and options. Finally, in Section 5, we present the results for some stable and working experiments, while Section 6 concludes our work.

2. TECHNICAL BACKGROUND

During the process of setting up the topologies to execute our experiments, we used a number of applications, technologies and software tools. Therefore, we deem it necessary to provide more information about them and report on documentation as this will be encountered by less experienced developers the first time they endeavour to delve into these technologies. Given that we were not complete experts ourselves with these technologies, this is largely what we had to start with, when we began this exploratory study.

2.1 Testbed Applications

In our work, we use two applications, *eBook Store* and *LEGIS* (Locality-Enhanced Geographic Information System) [9]. eBook Store is a homegrown application, which we have used extensively in our previous work as a representative example of a simple three-tier web application. LEGIS is a more sophisticated application, which follows a four-tier architecture with an additional big data analytics layer separated from the application's logic.

There are a number of reasons why we decided to use two applications in our study. The first reason is because we have extensive experience with eBook Store, which could help us set up the container cluster easier and interpret the results with more expertise. On the other hand, we wanted a more complex application to study more implications of the scaling process. LEGIS provides this complexity, since it uses distributed data analytics and it is based on a NoSQL database, compared to the relational database of eBook Store. This setting gives us a greater variety of more interesting scaling options.

2.1.1 eBook Store

eBook Store is a minimal three-tier web application [8]. An Apache load balancer sits at the front acting as the interface for accepting requests from clients, which are then distributed to the actual application. The application is hosted on a number of Tomcat web servers forming a scalable cluster. At the back-end of the topology lies a MySQL database. The application is a simple Java application which issues a number of different requests (select, insert, update) of variable intensity (i.e., number of records). We can control the type and intensity of the requests, which allows us to stress test specific resources (CPU, memory, disk, network) and activate particular scaling plans. The simplicity and degree of control over this application give us the flexibility to extensively test various conditions and scaling actions and devise a generalizable resource management strategy for most kinds of web systems. On the other hand, despite its simplicity, the architecture is popular and still relevant for most real-world applications, which gives us a satisfying level of applicability for the results of our study.

2.1.2 LEGIS

LEGIS was proposed and developed as an idea for the design challenge of the SAVI project [12]. It provides high throughput and low response time navigation services by analyzing local traffic conditions in real time. It draws its advantages from having its data and functionality partitioned in a geographically distributed multi-tier cloud platform. In such a platform, ”smart edges” constitute low capacity but high throughput cloud nodes, which can take up a large number of small tasks and the core acts as the high capacity datacenter, which can execute more demanding tasks and store larger amounts of data. The edges can be geographically distributed, which allows us to partition traffic creating responsibility zones. Within these zones, the edges accept navigation requests and analyze only their traffic data to calculate optimal travel paths. The edges communicate with each other through an internal high-speed network to aggregate the partial results into a single response for large paths that span across multiple zones. By partitioning the traffic data and localizing the analytics, we can achieve faster and more accurate results.

The architecture of LEGIS is more complex than eBook Store, as it introduces one additional layer of functionality. The Apache load balancer and the scalable Tomcat cluster are common between the two applications. LEGIS has an additional scalable analytics cluster based on Apache Spark. Finally, at the back-end of the topology, instead of a single-node MySQL database, we have a distributed NoSQL database cluster, based on Apache Cassandra. The application accepts navigation requests and a set of alternative paths is acquired from an external directions service. The results are passed to the Spark cluster which accesses Cassandra and based on the current traffic data annotates the path segments with a score, which determines the navigability of this particular path and its travel time. The annotated path is then returned to the client.

This application represents a more realistic example of modern data-intensive web applications, which allows for more complex scaling strategies. Additional challenges may rise with respect to resource management. We can control the intensity of the workload (based on the size of the paths) and we can trigger scaling in two clusters, Tomcat and Spark.

2.2 Apache Spark

RIGHTSLINK

Apache Spark[1] is a big data and distributed analytics service similar to Google's MapReduce [7] and Apache Hadoop [2]. Unlike these projects, however, Spark is more memory-intensive rather than disk-intensive. This allows to perform much faster analytics, with a potential tradeoff with respect to the size of the tasks. The Spark cluster consists of a master node and worker nodes. A cluster can be created[3] either directly in a standalone mode or using third-party cluster managers like Apache Mesos[4] or Hadoop YARN[5].

The master acts as a load balancer/gateway. It accepts job submissions from clients and then breaks it in tasks and distributes those to the workers. Jobs are submitted as bundled files (e.g., JAR files for Java jobs), along with their dependencies, i.e., libraries and drivers for connections with databases. The entire job file has to be sent to the workers, so that they are capable of executing the individual tasks. The master is responsible for marshalling the individual responses from the workers into a single response to be returned to the client. When a job is submitted a Driver program, containing the context (configuration) of the job, spawns Executor programs for the workers that will run the tasks of the job.

2.3 Apache Cassandra

Apache Cassandra[6] is widely used NoSQL wide-column data store, which is regarded highly by the database community [11, 13]. It is a distributed data store, which, as any NoSQL database, relaxes on ACID properties and guarantees availability with eventual consistency, which implies that the system will respond to any request but there is no guarantee that the response will be accurate or consistent at the time of the request. Unlike other wide-column NoSQL data stores, like Apache HBase[7] or Accumulo[8], Cassandra is *masterless*, implying that there is no master node in front of the workers. Cassandra works with a peer-based system, where all the nodes communicate with each other. Nevertheless, when a Cassandra cluster is set up, apart from the peers, there is also a seed, which acts as a master and knows all the peers in the cluster so that it can set up the connections between them.

2.4 Docker

Docker is an open-source project that aims to facilitate the development, deployment, delivery and execution of applications using containers. In practice, the mission of Docker is to implement to the best degree possible the *DevOps* paradigm [15, 5], which bridges the gap between development and operation management. In this paradigm, the developers focus only on building their software and not so much on managing it as there are tools, which automate this management as much as possible and also enable continuous deployment and fast release cycle.

Containers are packages that contain everything an application needs (binary files, dependencies, libraries, configuration files, etc.). This package, known as a Docker image, contains the application and its execution environment and makes it ready to be deployed in a container without further configuration. Images are reusable and extendible to allow for customizations. An application shipped as a Docker image will run on any Linux system (and recently on Microsoft Windows and Mac OS), regardless of what customizations has been made to it. Unlike VMs, a container does not contain an operation system, but rather inherits the host's OS to run. This way a container can be stripped of all files required for an operation system, and have only the files required by the application and nothing more. Because of this, the containers are lightweight, consume much less memory, and start much faster than a VM. At the same time they provide a high level of isolation (containers isolate applications from one another), thus providing additional security by default.

2.5 Scaling: The MAPE-K loop

The design and development of adaptive software systems usually follows the MAPE-K architecture [1]. According to this architecture an autonomic management system has four components for **M**onitoring, **A**nalysis, **P**lanning and **E**xecution. To achieve proactive or model-based adaptation a **K**nowledge base may also be part of the management system. Sensors, or monitoring agents, are deployed with the managed software system and are responsible for gathering performance or other measurements for the Monitoring component. These are then passed to the Analysis module, so that the health of the system is determined. If something is found to be out of order, specific scaling actions, e.g., scaling resources, are planned for particular situations. The planned actions are then passed to the Execution engine to be applied on the system. This process is repeated in frequent intervals to guarantee the system's quality and normal behavior.

Monitoring agents may gather measurements from multiple sources, including the application or the infrastructure, or even different components of the application. The complexity of the analysis may vary from simple IF-THEN rules to more complicated algorithms that are based on models. Planned actions include simple adding or removing resources in a static or dynamic manner or more sophisticated actions that would affect the application's configuration or other non-functional parameters. Finally, in cloud-deployed systems, the execution of adaptive actions can be the responsibility of an orchestration engine, like OpenStack's Heat, or a resource provisioning service.

2.6 DevOps

DevOps [10] can be considered as a novel development process or model or paradigm or even philosophy. Its primary goal is to bridge the gap between development and operation management. In this capacity, it recommends the use of tools and processes, as well as knowledge and skill sets, which would span across the entire lifecycle of the software. The concept existed, even before it was named DevOps [14]; developers would tinker with system administration tools and concepts to better understand how the software is to be deployed, while IT operators would occasionally merge with the development team to better understand the system's functionality and ensure higher quality. Thanks to virtual-

[1] http://spark.apache.org/
[2] http://hadoop.apache.org/
[3] http://spark.apache.org/docs/latest/cluster-overview.html
[4] http://mesos.apache.org/
[5] https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html
[6] http://cassandra.apache.org/
[7] https://hbase.apache.org/
[8] https://accumulo.apache.org/

RIGHTSLINK

ization technologies and the self-healing and self-adaptation capabilities of modern system, DevOps is the beginning of a new software development culture and a new breed of hybrid developers and IT specialists.

Another mission of the DevOps model is to shorten the release cycle of the software [16]. The goal is for every change to be incorporated seamlessly to the system in production, while high quality is ensured and maintained [4]. Netflix is already employing *chaos engineering* techniques [3], which, in the context of DevOps, enables concepts such as *continuous deployment* and *continuous delivery*.

Balalaie et al. [2] present their experience in migrating a monolithic mobile back end as a service (MBAAS) to a microservice architecture. Apart from the common pains and suffering of migrating a legacy system to a service-oriented architectures, they had some interesting experience with respect to the DevOps side of the process and the final system. First, they had to change the team organization from a horizontal structure (development, QA, operations) to more vertical teams with all layers responsible for the smaller services. Second, they report on the importance of monitoring the system and, finally, on the use of containers to bridge the gap between the development and the production phases.

3. IMPLEMENTATION OVERVIEW

Given our goal to study the DevOps aspects of multi-layer and multi-tier data-intensive web applications, we started our experiments with LEGIS on Docker containers. LEGIS was developed prior to this study, but it was not fully profiled. Therefore, while we had a general idea about its demands, we did not have quantitative data about the application's performance. As a result we knew roughly what we will need for the system in terms of resources, but we did not know what we were to expect in terms of the application's performance and eventually about its scaling needs.

3.1 Docker Deployment

To deploy LEGIS on containers, we first had to set up the Docker cluster. Installing the Docker Engine, the provisioning service of Docker, on a single host is a straightforward process. For example, to install it on Linux Ubuntu 16.04, it is sufficient to run the command `apt-get install docker.io` as root, and a functional engine will be deployed. However, to take full advantage of the power of Docker, it is recommended to be installed on multiple hosts in a cluster. The resources available in those hosts (CPU, memory, disk) will form a pool from which the containers will take as necessary. However, when a container starts it will take *all resources it needs from a single host*, the host it resides in (i.e., a container that requests 2GB RAM won't be able to take 1GB from one host and 1GB from another one); if no single host can satisfy the requirements, the container creation fails.

In addition to the provisioning service, we also need a cluster manager. Docker Swarm[9] is the project that offers native clustering functionality for Docker. The project aims to transform a set of multiple Docker hosts into one large *virtual* Docker host. In our deployment, one VM was commissioned as the *cluster manager* and hosted services critical for the function of the Swarm Cluster, and nothing else. The rationale is that we wanted to isolate the cluster management node from the worker nodes. In the case that the workers receive enough workload to saturate the VM, the manager should remain unaffected. This decision was necessary and important, since the goal of our experiments was to create saturation and scaling scenarios. The rest of the VMs in the cluster would assume the role of Swarm Nodes and would play host for containers.

[9]https://docs.docker.com/swarm/

The Discovery Service.

In order to keep track of cluster members (Docker Engines that are joined to the cluster) and their IPs, Docker Swarm makes use of a discovery service. In our experiments we have used Consul[10], but ZooKeeper[11] and Etcd[12] can also be used. We installed Consul on the *manager VM*, and started the agent in server mode with the command `consul agent -server -bootstrap`. An important element to notice here is the `-bootstrap` parameter, which tells Consul to start in single mode and not wait other Consul servers. In a distributed, highly dynamic environment it is recommended to run multiple Consul servers; if one of them becomes inaccessible (crashes, network or hosts problems), the discovery service can still be accessed through the other servers. This means that the service provides *High Availability* (HA), an important feature in distributed environments. In our experiments the whole deployment was relatively small (no more than 10 VMs for all services), and we have disabled the HA functionality. The `-bootstrap` parameters tells the server not to run its *consensus protocol* with the other servers, and assume it will be the only one.

The overlay network.

Because the containers that will run in the cluster could end up on multiple hosts, they need to connect to each other and can do so only through the IPs of their hosts. As a consequence, a container that runs Spark and wants to register with the Spark Master container, must know the IP of the host where the Spark Master is located. In addition, because on a host a port can be used only by one container and there can be multiple containers running on the same host, the ports used by containers cannot be known beforehand. For example, the creation of a container that asks for port 80 by specifying the parameter `-p 80:80` could fail, if another container has port 80 in use. Therefore, the creation of the container must instead use the parameter `-p 80`, letting the Docker Engine to choose a random port on the host to which to bind port 80 from the container.

Spark Swarm offers a better solution to allow containers to connect to each other, *overlay networks*. An overlay network, with a particular `<name>`, can be created using the command `docker -H :4000 network create --driver overlay --subnet=192.168.100.0/24 <name>`. When starting a container, it can be attached to this network and receive an IP on it by specifying the parameter `--net <name>`. If the container also specifies a hostname (using the parameter `--hostname`, e.g., `--hostname spark-master`), then it can also be accessed using simply the hostname. For example, if there is a container on the overlay network with the hostname `spark-master` that runs the Spark Master process on the default port (7077), a Spark Worker could connect

[10]https://www.consul.io/
[11]https://zookeeper.apache.org/
[12]https://coreos.com/etcd/

RIGHTSLINK

to it using the address `spark-master:7077`. The hostname would be resolved by the DNS service provided by Swarm.

Docker images.

For our testbed applications we created custom Docker images for all potential containers. These included a Tomcat image with Java installed and the two applications deployed on the server, an Apache2 load balancer image, a Spark image with scripts to start either the master or the worker services and the spark job for LEGIS, a Cassandra image with scripts to load the traffic data and a MySQL image for the eBook Store. Some of these images like the load balancer or Tomcat can be used for both applications.

3.2 Application Deployment

Concerning LEGIS itself, we decided to simplify its structure and its functionality slightly to allow for more controlled experiments. To this end, we deployed the service only in one cloud region, with travel paths and traffic data only for this region. We also used a precalculated path to avoid issuing requests to the external directions service to minimize potential third-party failures. As a result, there is an existing JSON file with the path, which is forwarded to the Spark cluster for annotation. Furthermore, we reduced the size of the original path to include only a dozen segments for annotation to reduce the calculation time and allow for higher throughput and faster experiments. Finally, we preloaded the Cassandra data store with 2-hour worth of traffic data for this particular data from the CVST project [6], which gathers such data from the Greater Toronto Area in Canada.

There were no special considerations for migrating eBook Store from VMs to containers and as soon as the images were ready, the application was deployed as in our previous experiments [8].

4. PROCESS AND EXPERIENCE

Having set up the Docker Swarm cluster with virtual machines and having prepared all the Docker images, we deployed the LEGIS application on containers. Then, we initiated a simple workload generator, which simulates a constantly increasing number of clients issuing requests to the LEGIS service. After a few iterations, we started encountering the first problems.

4.1 Deployment and Delivery

Challenge 4.1.1 (Increasing Memory Utilization)**.** We noticed that memory utilization in both the Tomcat and Spark clusters was increasing constantly, almost linearly, even after requests were served and the clients received the respective responses.

Option 4.1.1.1 (Cleanup and More Efficient Memory Utilization). Our first action was to refactor both the web application and the Spark job to make sure that there were no memory leaks and that the two pieces of software generally handled memory in an efficient manner. After the refactoring, we managed to significantly reduce memory consumption on both clusters, but the utilization kept increasing steadily.

Option 4.1.1.2 (Multiple JVMs on Tomcat). Focusing on Tomcat, we found that these particular containers reported excessive memory consumption due to the fact that the web application submitted Spark jobs through `spark-shell` and for every request an new job was submitted and as a result a new Java process was created. Therefore, we decided walk around this submission process, by having the job (the bundle JAR file) already deployed in Spark and be able to invoke the submission remotely via a REST API. We found that Spark has indeed such an API, but it is hidden. This implies that the API may not be fully tested or stable and, as such, it is not part of the official documentation of Spark. Nevertheless, we decided to use the API to address the multiple JVM problem. Indeed, the problem on Tomcat was solved and memory consumption was regulated. However, in Spark the problem persisted.

Option 4.1.1.3 (Zombie Processes). Zombie processes are OS processes that finished their execution, but the result code has not yet been consumed by the parent process. Zombie processes occupy system resources (i.e., memory) that need to be released. On a Linux system, the `init` process removes the zombie processes, making sure to consume their exit result even if their parent does not do so, thus releasing all resources associated with it. In a container, where there is no full operating system running, and the `init` process does not exist by default, there is the risk of ending up with zombie processes. It is the responsibility of the developer to make sure that all processes inside the container terminate properly and release all the resources when they finish. Eventually, after making the appropriate changes to the Spark job, we fully addressed the problem.

Challenge 4.1.2 (Spark Jobs High Execution Time)**.** Focusing on the Spark cluster, we noticed that it took jobs too much time to finish, in the range of minutes. The simplicity of our analytics, the fact that by default we had at least two Spark workers and not too much workload yet did not support the long execution time.

Option 4.1.2.1 (Network Issues). Having fixed the memory issues and given the fact that our application did not have any I/O operations and there was not much CPU utilization, as our monitoring indicated, the only remaining source for a bottleneck was network. Upon closer inspection we saw that when an application submits a job, the whole bundle JAR file is transmitted from the Master to all the workers. As it happens, our job file was rather large due to a large number of library dependencies. To solve the issue, we removed all these libraries and load them in the Spark containers upon creation. We also changed the Tomcat context on the application side, so that it passes the dependencies' location to Spark job when this is submitted. This way we managed to reduce the execution time to a few seconds, which agreed with our measurements, when we executed the job locally.

Challenge 4.1.3 (Cassandra Crashes)**.** During the first experiments, we had Cassandra failing a few times. At the same time, we also observed disk saturation to some Docker hosts. Even though the two events were observed independently, due to their temporal and spatial coincidence (the saturated VMs were the ones that hosted Cassandra containers) led us to point out the correlation. When we focused on this problem, we actually noticed that Cassandra was collateral damage rather than the culprit. The real reason for the disk saturation was Spark, whose containers were sharing the same host with Cassandra and which, having limited memory resources were relying on disk to complete their tasks. This did not allow enough disk space for Cassandra, which failed, causing also Spark workers to freeze.

RIGHTSLINK

Option 4.1.3.1 (Keep Spark and Cassandra Separate). The first solution to this problem is to actually make sure that Cassandra and Spark containers do not end up in the same host.

Option 4.1.3.2 (Keep Cassandra in VMs). Unfortunately, when Cassandra containers crashed, data got lost and had to be reloaded when the containers was restarted. This prompted us to take a more permanent solution and take Cassandra out of the swarm cluster in an independent and persistent VM. We also refrained from executing any adaptation actions on Cassandra for fear of additional failures.

Challenge 4.1.4 (Containers Crash Without Trace)**.** Thanks to the experience with Cassandra and Spark containers failing, we noticed that when this happens, the container is no longer accessible, which means that we have no access to the deployed software log files either. This creates particular problems when debugging the failure.

Option 4.1.4.1 (Write Logs to Mounted Volume). Logs can be written to files directly in the host VM by mounting the target folder as a volume to the container. This way log files will persist even after the container fails. However, mounted volumes generally carry security risks, due to containers sharing hosts, and, thus, additional attention needs to be paid in this case.

Option 4.1.4.2 (Main and Secondary Processes). We noticed that a container fails when its main process fails. Having our deployed software (Tomcat, Spark, Cassandra) as the main process, when something went wrong in our experiments, the entire container would fail. For this reason, we found it more prudent to start containers with another program as the main process (e.g., `bash`) and add our software as another process. This way we can still access the container even after our software fails, and be able to inspect the failure.

Challenge 4.1.5 (Multiple Network Interfaces)**.** Some applications, like Apache Spark, require to bind to a specific network interface (NIC) and not to the generic 0.0.0.0 which would enable to receive connections on any interface. If the container has multiple NICs (which is not uncommon, considering that one NIC is the `bridge` to the host and another one is used by the virtual overlay network), then, at container startup, the correct binding IP address must be known inside the container. The exact address cannot be known before the container starts and receives one from the Docker Engine.

Option 4.1.5.1 (Environmental Variables). A possible solution is to use environmental variables (`-e` flag for the `docker run` command) to pass information inside the container, which would be used in scripts to extract the right address. Passing the name of the network interface (i.e., `eth0`) is challenging because the names of interfaces are not always the same (it is possible that in a container with multiple NICs, none of them is named `eth0`).

Option 4.1.5.2 (Inspect Container). Another solution is to run inside the container the command `docker inspect ...`, if the file `/var/run/docker.sock` from the host is shared with the container, and find the container address in the response. This, however, requires that the docker engine is installed inside the container.

4.2 Management and Scaling

Moving from deployment to management, we realized that we are working with two new concepts, for which we were not overly familiar, Docker and Spark. To this end, we were not always certain about the source of some of our problems. Therefore, we decided to divide the work and, on one hand, work with eBook Store on containers, for which we knew the details of the application, and on the other hand, with LEGIS on VMs, which had not tried before. Eventually, we were able to also deploy LEGIS on containers.

Deploying eBook Store in containers did not present any particular challenges, mainly because we knew all the details about the application and we had previously addressed most deployment problems during our LEGIS work. New problems started rising after we began to stress test the application with intense workload, both in frequency and in size in terms of number of clients. Stress testing would eventually create saturation points, which will require adaptation and scaling.

In order to enable autoscaling for the containerized topology of eBook Store, we had to implement the MAPE-K loop. We considered it unnecessary to focus on sophisticated analysis and planning components, because, first, simpler methods are already in use by practitioners in actual projects, and, second, our intention was to see if and how is scaling itself possible in such topologies. In addition, execution was simple enough to take care of, thanks to Docker commands and the Swarm manager, which allowed us to add containers to the cluster given an image.

Challenge 4.2.1 (Monitoring Containers)**.** Eventually, what remained to be explored was monitoring of containers. Docker provides a REST API to access monitored metrics for containers. The API returns metrics for CPU, memory and bandwidth consumption for a specific containers. However the metrics were not as straightforward as the ones returned by tools operating on the VM level and with which we were more familiar, like Ceilometer for OpenStack or CloudWatch for Amazon EC2. In addition, the official Docker documentation on monitoring focuses mostly on how to access the API for a specific container, but without adequate details on how to interpret the response.

Option 4.2.1.1 (Interpreting Metrics from Docker Stats). Eventually, after looking into side documentation sources, like forums or StackOverflow (including the source code repository of Docker), we found that the Docker API returns measurements of pure consumption in raw format, i.e., bytes of memory or I/O and nanoseconds of CPU time. Given this raw format, we have to transform the measurements in percentages to express utilization. The API reports the last measured metric and the current measurement, which was requested. To get the final metric, we normalize the difference between the two measurements over the "system available". Formally,

$$Cpu\% = 100 \times \frac{ContainerCpu_{cur} - ContainerCpu_{pre}}{SystemCpu_{cur} - SystemCpu_{pre}}$$

Challenge 4.2.2 (Inconsistencies in Monitoring)**.** Having started our workload generator, which progressively increased its intensity, and while reading the measurements from monitoring both host VMs and containers, we made certain observations that did not make full sense. Most importantly, we saw the response time of the application climbing to high levels, while the containers and the VMs did not report any saturation on any metric (CPU or memory).

RIGHTSLINK

Option 4.2.2.1 (Metrics per Core or per VM). Looking into the Docker source code revealed another fact about monitoring; the measurements pertaining to CPU are in fact cumulative for all the cores of the host VM. Therefore, for example, if the host has 4 cores, the measurements range from 0 to 400%. This misinterpretation led us to present wrong results and prevented us from predicting the saturation of the containers. By correcting the normalization step of the calculations fixed this issue.

Challenge 4.2.3 (No Effect of Scaling)**.** In the first experiments with eBook Store, when the topology became saturated and the scaling actions were triggered, we noticed that the actions had no effect to the application's performance and the containers remained saturated. When we investigated the problem, we understood that when Docker deploys a container on an empty VM host, the container takes all available resources. When a second container is deployed in the same VM, the resources are simply split between the two containers. If the host is already saturated, the new container will take very limited or no resources at all, rendering the scaling action mute.

Option 4.2.3.1 (Scaling VMs). In this scenario, upon saturation and before the new container is added, it is better to scale the host VM either up (i.e., increase its size) or out (i.e., add a new host VM in the Swarm cluster). However, having to scale the VM cluster removes any obvious advantage of using containers with respect to managing resources efficiently and in a cost-effective manner.

Option 4.2.3.2 (Putting Restrictions on Container Resources). To better alleviate this situation and better design the management of containers, we decided upon restricting the resource allocation to containers. Docker gives this option when creating and starting a container. In practice, this creates "blocks" or "units" of deployment. The expectation is that adding these blocks in the host VMs as part of the scaling process will indeed have a positive effect on the performance of the application. In this context, a host VM or a Swarm cluster will become saturated once they run out of space to host additional blocks. This creates the notion of *capacity* for the VMs in terms of containers, simplifying the concept of scaling and, thus, bringing us closer to the DevOps paradigm.

Challenge 4.2.4 (Restrictions on Container Resources)**.** Deciding upon restricting containers is in fact easier than actually implementing it. In our experiments, there are two kinds of restrictions that we have applied to containers: memory and CPU. Memory is more straightforward and poses no particular challenges: out of the total amount of memory available in a host (e.g., 8GB in our case), we allocated to the container only a fraction (1GB). The container would not be able to use more when it used its quota and would show signs of saturation. On the other hand, CPU restrictions are more challenging as in practice they require the implementation of a *timeshare* between the containers on the CPU capacity of the host.

Option 4.2.4.1 (Imposing CPU Restrictions on Containers). For CPU restrictions, we have used the `--cpu-period` and `--cpu-quota` parameters when starting a container. These two parameters, together, influence the *Completely Fair Scheduler* used by the host operating system to allocate CPU time to containers. Although the containers are managed by the Docker Engine, they essentially run as processes on the host operating system, making use of the *cgroups*—or *control groups*—feature of a Linux kernel. The two parameters are sent directly to the operating system when such a process is created. The intuitive way to understand these parameters is *for every* `cpu-period` *microseconds that pass, the container is allowed to use only* `cpu-quota` *microseconds.* If the container requires more than its allocated share, then it will be *throttled* by the operating system. It is important to note here, that the access to resources—memory, CPU—is managed directly by the host's operating system, and not by the Docker Engine.

This interpretation is straightforward when the host has only one vCPU, i.e., virtual core. If the host has multiple cores, For example 4, then for every `cpu-period` microseconds, there are 4×`cpu-period` microseconds of usable time on the CPU (one `cpu-period` for each core). Out of this 4×`cpu-period` microseconds, the container is allowed to use only `cpu-quota` microseconds.

If the container is configured to use the entire CPU of the host (i.e., `cpu-period` is 1000 and `cpu-quota` is 4000), then the container will become saturated when it reaches 400% host CPU utilization. If the container is configured with `cpu-period` set to 1000 and `cpu-quota` set to 1000, then the saturation will be reached when the host CPU utilization is at 100%.

Challenge 4.2.5 (Saturation of the DNS Service)**.** After a number of experiments with eBook Store and successful scaling of the container topology, we suddenly noticed that response time was going up, without containers reporting any kind of saturation, and the arrival rate, i.e., the number of requests received by the application per second from all users, was capped at approximately 80 requests per second. Naturally, one would expect that with the response time growing, the arrival rate to decrease due to the users lack of ability to issue requests faster. This suspicious behavior led us to further investigate the issue and we found that it was the Docker DNS service that was getting saturated. The problem was that all the communication between containers was conducted based on hostname. Therefore, for every connection to be established, Docker had to look into the DNS service, find the corresponding IP and then establish the connection. Apparently, the service could only accept upto 80 concurrent requests per second. The phenomenon was more evident early on in the load balancer, which had to forward the incoming requests to Tomcat servers. This was confirmed by the increase of CPU utilization of the load balancer (under normal circumstances, the load balancer takes insignificant load).

Option 4.2.5.1 (Scale Up the Swarm Manager). One possible solution is to increase the size of the VM that hosts the Swarm Manager, which holds the DNS service. In our experiments, we increased the size of the Manager host from medium to large, which allowed to surpass the cap of the 80 requests per second. However, the cap reappeared a little bit later at 100 requests per second, which showed that this solution can at best be a temporary fix.

Option 4.2.5.2 (Use IPs instead of Container Names). The more prudent option is to use directly the IPs of the containers instead of the names and completely bypass the DNS service. Nevertheless this is not trivial, mainly due to the overlay network. IPs may not be known at design time, or

RIGHTSLINK
Copyright Clearance Center

even at scaling time, which may result in delays or even worse in non-established connections.

Option 4.2.5.3 (Cap the Arrival Rate). Eventually, for our experiments, we decided upon another temporary fix, which was to make sure that the arrival rate, as it comes from the workload generator, does not reach the cap set by the DNS service. At the same time, we increased the demand of requests for CPU to achieve the desired saturation conditions earlier on.

Challenge 4.2.6 (Spark Driver-Executor Deadlock)**.** Having a relatively successful scaling experiment for eBook Store, we moved on to work with LEGIS. With all the deployment problems fixed, we managed to get a few iterations in the LEGIS experiments with some scaling as well. After that, we experienced a similar problem to the DNS cap from the previous experiments; requests stopped getting accepted after some point (and in fact were decreasing with time), without any indication of saturation. Identifying the source of this problem was easier in this case. The Spark Master graphical interface informed that the problem was in fact with Spark and not with Docker. When a job is submitted a Driver process is created to manage the individual tasks that will be handled by the Spark workers. A Driver takes up one CPU core and some memory, as indicated in the configuration of Spark. The Driver will spawn Executor processes for every task in a Spark worker. The Executors also require one core and memory. In the case that many requests arrive in the system and many Drivers need to be created, we end up in a deadlock; Spark creates all the possible Drivers, which take up all the available resources, leaving nothing for Executors to be started. At this point, it seems that Spark does not do resource allocation in an optimal manner, as it gives priority to all managing processes, i.e., Drivers, and not to the processing ones, i.e., Executors, which upon termination will release all resources. Given the cyclic dependence between the two processes (an Executor needs a Driver to start, but a Driver needs the Executor to finish first), Spark should make sure to start the entire pair, before starting to serve new requests.

Option 4.2.6.1 (Scale Up Host VMs for Spark). One straightforward solution to problem is to increase the size of the host VMs and the restrictions on the containers that hold the Spark processes. This will make more resources (vCPU and memory) to become available for Spark jobs. However, as in the first option for the DNS cap problem, this is a temporary fix and the problem will recur once the arrival rate is further increased.

Option 4.2.6.2 (Better Memory Monitoring). Eventually, what needs to be done is better resource management. CPU cores are not necessarily a significant problem, because for Spark cores mainly represent *threads*, which means that upon submitting the job we can inform Spark that there is an arbitrarily large number of cores available in the host container or VM. This transfers the problem how many connections the service can accept, but this number is usually larger than the number of cores and it can handle a large enough arrival rate. Memory, on the other hand, has to correspond to the actual memory of the host resource. For this problem, we need to have handles, which make sure that there will be enough memory to spawn at least one Executor before spawning a new Driver. With at least one Executor in the queue, this means that once the process finishes, resources for two processes are released and we can spawn new ones, at least one of which will have to be an Executor.

5. EXPERIMENTAL RESULTS

Throughout our study we conducted a large number of experiments varying in duration and settings, which helped us identify the challenges and evaluate the options we reported previously. Eventually, we narrowed it down to four experiments (or sets of experiments), in which we were able to clearly demonstrate autoscaling capabilities for containers. Two experiments concerned the eBook Store application; one with linearly increasing workload on Amazon EC2 and one with variable workload on SAVI [12], an OpenStack research cloud. Another two experiments were conducted with LEGIS on SAVI with linearly increasing workload, one on VMs and another on containers.

Figure 1 shows the results for eBook Store on Amazon EC2 with linearly increasing workload. The top plot shows the CPU utilization of the two Swarm hosts (worker 1 and worker 2). The second plot shows the CPU utilization of the containers (for MySQL, load balancer and the average for the Tomcat web workers). The bottom plot shows the arrival rate in requests per second and the response time in seconds. The two halves of the plots correspond to different settings with respect to resource restrictions on containers. In this case, we define 1 unit of computation (C) (i.e., one container block) having 12.5% CPU out of 200% (for two cores) and 512MB RAM out of the total 8GB of the host. We configure the load balancer and the MySQL to take 4C each and the Tomcat workers 1C each. With two Swarm hosts and by putting the load balancer and the database in the first, this means we have capacity for 24 web workers; 8 in the first host and 16 in the second. In the right half of the plot, the configuration is 6C for the load balancer and the database, which leaves 20C for web workers, 4C in the first host and 16C in the second host.

The results clearly show that, when we reach the capacity of the hosts (24 and 20 of web containers respectively for the two halves), we also saturate the host VMs in terms of CPU utilization. Additionally, we see that scaling the web layer, we managed to maintain the average CPU utilization within the desired range, between 60% and 80%. The leftmost plot shows that there is an early saturation of the load balancer and of the database, which prevented us from issuing more requests. When we increased the allocated capacity to the corresponding containers (from 4C to 6C), we were able to achieve higher throughput until the web workers were saturated and at which point, that we reached its container capacity, the Swarm host was also saturated.

Figure 2 shows the results for eBook Store on the SAVI cloud with variable workload. In this experiment, the restrictions are defined so that 1C has 1GB RAM out of 8GB of the host and 12.5% CPU out of 400% (for 4 cores). The load balancer has 6C, MySQL has 4C and each Tomcat server takes 1C, as previously. For three Swarm workers, this leaves capacity for 14 Tomcat workers in total. The CPU utilization thresholds for the Tomcat workers were 60% and 90% respectively. Given that the workload does not increase infinitely in this scenario, we never reached the hosts' capacity in terms of containers. However, our reactive autoscaling works fine, maintaining the web workers' utilization within range and, as the plots show, the number of containers smoothly follows the arrival rate.

RIGHTSLINK

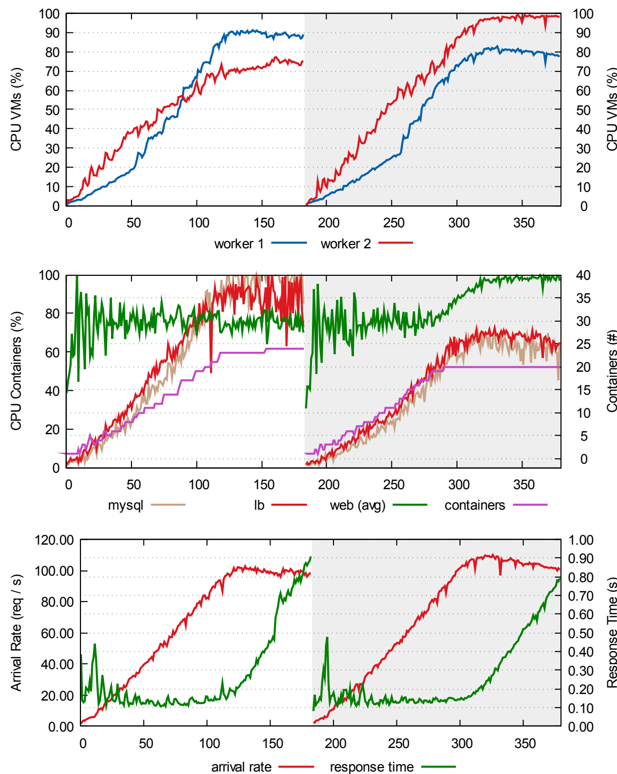

Figure 1: eBook Store on Amazon EC2 scaled on containers with linear workload.

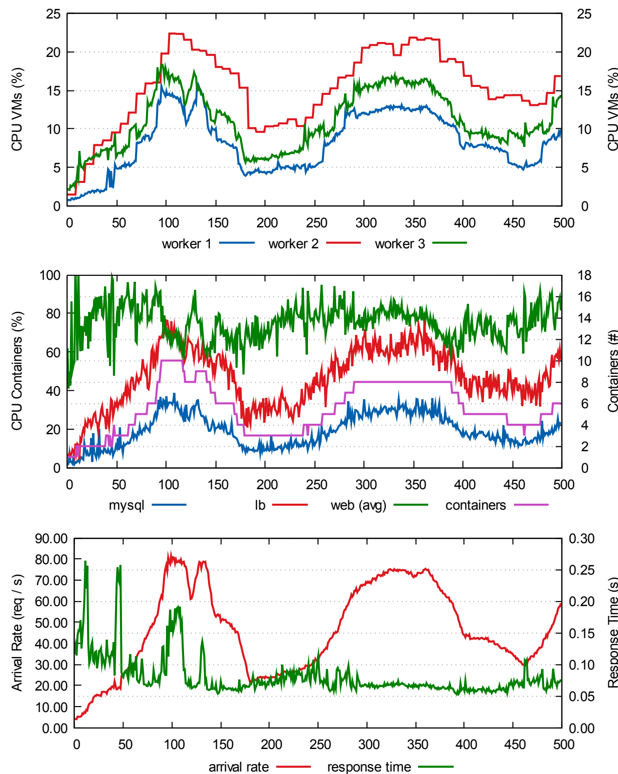

Figure 2: eBook Store on SAVI scaled on containers with variable workload.

Figure 3 shows the results for LEGIS on VMs. The purpose of this experiment was to focus only on Spark, without Docker, to see if we are able to activate autoscaling and what would be the effect of scaling multiple tiers. The results show that we are capable of multi-tier autoscaling. The process managed to maintain average CPU utilization for both layers within range, which, for this experiment, was set between 30% and 70% of CPU utilization.

Finally, Figure 4 shows the results for our ultimate goal, scaling a multi-tier data-intensive application, LEGIS, on containers. A linearly increasing workload was used in this experiment, while the unit of computation was defined as having 256MB RAM out of 8GB in the host and 12.5% CPU out of 400% (4 cores). The load balancer, Spark master and each of the Spark workers were assigned 4C, while Tomcat workers were given 1C each. In three Swarm hosts, this meant we had a capacity between 20 Spark worker containers and 2 Tomcat workers to 1 Spark worker and 80 Tomcat workers. The utilization thresholds were set between 60% and 80% for Tomcat workers and between 40% and 80% for Spark workers. The results show good scaling capabilities to accommodate an increasing workload, especially on the Spark tier, good maintenance of the response time, until saturation was reached on the VM that hosted the Spark workers. On the negative side, we observe oscillation with respect to CPU utilization, but this mostly due to the nature of the application (longer tasks with higher intensity).

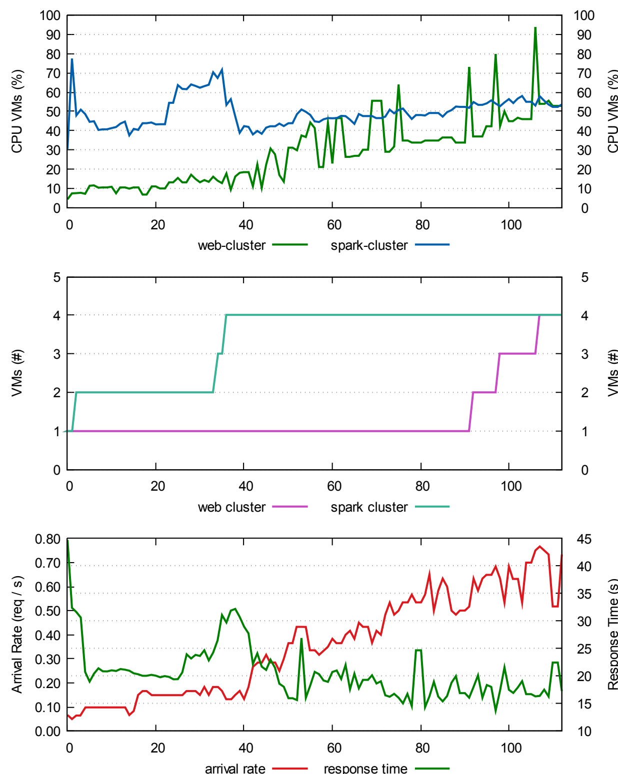

Figure 3: LEGIS scaled on VMs.

6. CONCLUSIONS

In this work, we reported our experience with developing, deploying and eventually managing data-intensive applica-

RIGHTSLINK

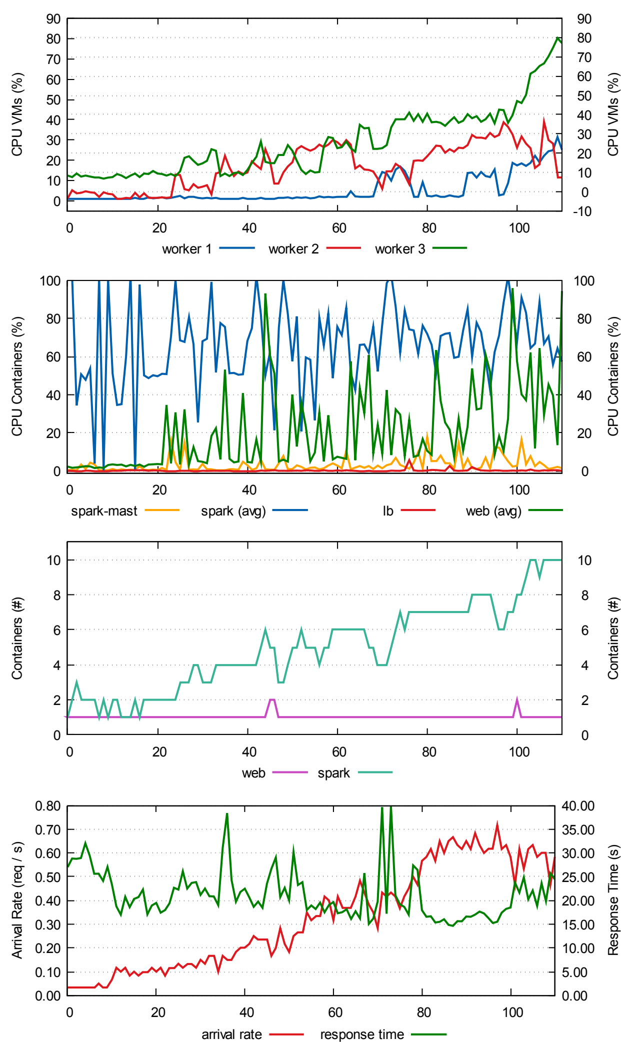

Figure 4: LEGIS scaled on containers.

tions on container-based cloud infrastructures. We focused on two application, a simple three-tier one with Tomcat and MySQL and a more complex four-tier one with Tomcat, Spark and Cassandra. The goal of our study was to explore and identify the challenges and pitfalls of these processes and eventually present stable and working settings for achieving autoscaling with containers for such web applications. Our experiments show that the decisions we made during the exploratory part of our study led to an appropriate and meaningful autonomic management system that operates on multiple layers (containers and VMs) for complex applications with more than three tiers of functionality.

Our study required significant effort and time before we were able to accomplish the desired experiments. In addition, we had to go through unconventional channels, including developer forums, source code repositories and Q&A sites, to find answers to several of our problems, especially concerning monitoring and putting restrictions on containers, which lead us to believe that Docker does not have detailed enough documentation, especially for developers not familiar with the technology. Moreover, to the best of our knowledge, we are one of the first research teams to stress test Spark clusters and Docker containers, although more with respect to throughput rather than intensity of analytics jobs, which revealed several limitations of the two technologies. Finally, for every problem we encountered, we considered a set of options and we chose the one that would progress our work in an expedite manner to allow us to run our experiments. At no point do we claim that we picked the best possible solution, or that we considered all possible solutions. This is an ongoing work and, with the accumulated experience from this study, we expect to continue improving upon our tools and methods. Eventually, we anticipate that this work will serve as a useful guide for practitioners and researcher, who embark on the development of such data-intensive containerized web applications.

7. ACKNOWLEDGEMENTS

This research was supported by IBM Centres for Advanced Studies (CAS), the Natural Sciences and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network and the Ontario Research Fund for Research Excellence under the Connected Vehicles and Smart Transportation(CVST) project.

8. REFERENCES

[1] An architectural blueprint for autonomic computing. Technical report, IBM, 2005.

[2] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[3] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[4] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[5] M. Bentley, J. DeVita, and V. Will. Making the Transition to DevOps. Technical report, Docker, 2015.

[6] CVST. Connected Vehicles and Smart Transportations. http://www.cvst.ca/, 2014.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] M. Fokaefs, C. Barna, and M. Litoiu. Economics-driven resource scalability on the cloud. In *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 129–139. ACM, 2016.

[9] M. Fokaefs, D. Serrano, R. Veleda, and M. Litoiu. Locality-Enhanced Geographic Information System. In *4th International IBM Cloud Academy Conference*, 2016.

[10] M. Hüttermann. *DevOps for developers*. Apress, 2012.

[11] H. Khazaei, M. Fokaefs, S. Zareian, N. Beigi-Mohammadi, B. Ramprasad, M. Shtern, P. Gaikwad, and M. Litoiu. How do i choose the right nosql solution? a comprehensive theoretical and experimental survey. *Submitted to Journal of Big Data and Information Analytics (BDIA)*, 2015.

[12] SAVI. Cloud platform. http://www.savinetwork.ca, June 2015.

[13] Solid IT. Knowledge base of relational and nosql database management systems, June 2015.

RIGHTSLINK

[14] D. Spinellis. Being a devops developer. *IEEE Software*, 33(3):4–5, 2016.
[15] J. Willis. Docker and the Three Ways of DevOps. Technical report, Docker, 2015.
[16] L. Zhu, L. Bass, and G. Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016.

RIGHTSLINK