Containerized A/B Testing

ÁDÁM RÉVÉSZ and NORBERT PATAKI, Eötvös Loránd University, Faculty of Informatics

Software version ranking plays an important role in improved user experience and software quality. A/B testing is technique to distinguish between the popularity and usability of two quite similar versions (A and B) of a product, marketing strategy, search ad, etc. It is a kind of two-sample hypothesis testing, used in the field of statistics. This controlled experiment can evaluate user engagement or satisfaction with a new service, feature, or product. A/B testing is typically used in evaluation of user-experience design in software technology. DevOps is an emerging software methodology in which the development and operations are not independent processes, they affect each other. DevOps emphasizes the usage of virtualization technologies (e.g. containers). Docker is widely-used technology for containerization. In this paper we deal with a new approach for A/B testing via Docker containers. This approach is DevOps-style A/B testing because after the evaluation the better version remains in production.

Categories and Subject Descriptors: K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software selection*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Evaluation / methodology*; D.2.9 [**Software Engineering**]: Management—*Software Management*

General Terms: Software Quality Analysis with Monitoring

Additional Key Words and Phrases: Docker, containers, DevOps, A/B testing

1. INTRODUCTION

Nowadays A/B testing plays an important role in evaluation of different but very akin user-experience design. It is widely used among online websites, including social network sites such as Facebook, LinkedIn, and Twitter to make data-driven decisions [Xu et al. 2015]. A/B testing is an important method regarding how social media affects the software engineering [Storey et al. 2010]. A/B testing has been applied in many web portals successfully [Kohavi et al. 2009].

A/B testing is a powerful method because it is based on the behavior of the end-users. A/B testing of webpages or webapplications requires small changes in the user-experience design (e.g. colors, structure of a page, shape of buttons, etc.). As visitors are served either the control or variation, their engagement with each experience is measured and collected. It can be determined whether changing the experience had a positive, negative, or no effect on visitor behavior from the collected info. A/B testing of these applications takes time for collecting proper number of feedback.

Containerization is new directive in virtualization: this lighweight approach supports operating system-based isolation among different pieces of the application [Soltesz et al. 2007].Containerization is on the crest of a wave since Docker has been developed. Docker provides a systematic way to automate the faster deployment of Linux applications inside portable containers. Basically, Docker extends casual Linux containers (LXC) with a kernel-and application-level API for improved isolation [Bernstein 2014]. Docker is emerging tool to start complex application in many virtual operating

This work is supported by EFOP-3.6.2-16-2017-00013.
Author's address: Á. Révész, N. Pataki, Eötvös Loránd University, Faculty of Informatics, Department of Programming Languages and Compilers, Pázmány Péter sétány 1/C, Budapest, Hungary, H-1117 email: reveszadam@gmail.com, patakino@elte.hu

Copyright ©by the paper's authors. Copying permitted only for private and academic purposes.
In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

system-separated parts and it ensures the communication among them. Docker has a comprehensive documentation [Docker Inc. 2017].

Docker containers are built up from base images, there are general images (e.g. Ubuntu 16.04) and specific images (for Python run environment). Dockerfiles describe how an image can be created and Docker is able to generate the image and save it to the repository. Many services of the Docker platform are available (e.g. Docker Engine, Docker Compose, etc.). The images are in Docker Registries. The Docker Engine is responsible for managing containers (starting, stopping, etc.), while Docker Compose is responsible for the configurations of containers on a single host system. Docker Compose is mainly used in development and testing environments. One can define which services are required and what are their configuration in the container. Docker Compose files can be created for this purpose. Orchestration of Docker containers are typically executed with Kubernetes or OpenShift [Vohra 2017].

Continuous Delivery (CD) is a software development discipline. This methodology aims at building software in such a way that the software can be released to production at any time. It is a series of processes that aims at safe and rapid deployment to the production. Every change is being delivered to a production-like environment called staging environment. Rigorous automated testing ensures that the business applications and service work as expected. Every change had been tested in staging, so the application can be deployed to production safely.

The DevOps approach extends the CD discipline and focuses on comprehensive CD pipelines: starts with building and followed by different kinds of testing [Schaefer et al. 2013]. Unit testing, component testing, integration testing, end-to-end testing, performance testing, etc. should be performed on the software [Roche 2013]. In the meantime, static analyser tools try to find bugs, code smells and memory leaks in the source code. 3rd-party compliance should be checked in the build pipeline. Automated vulnerability scanning of the software is mandatory to discover security gaps. The visibility of the whole process is guaranteed.

After this phase, the automatic deployment of application starts. Application Release Automation (ARA) tools are available that can communicate with the CI server and the deployment steps can be designed on a graphical user interface of these tools. The DevOps culture argues for the deployment automation at the level of the application [Cukier 2013]. The automatic upgrade and roll-back processes involve many difficult changes. Database schemas, configuration files and parameters, APIs, 3rd-party components (e.g. message queues) may be changed when a new software version is released. The deployment process has to cover these changes as well and requires automation and visibility.

DevOps considers the monitoring and logging of the deployed application in the production environment [Lwakatare et al. 2015]. The development team is eager for feedback from the application which is in the production environment: e.g. what are the unused features in the software, memory or other resource leak detection or performance bottlenecks. ELK-stack is a popular toolset for this purpose [Lahmadi and Beck 2015]. Elasticsearch is a distributed search and analytics engine, Logstash is a data processing pipeline and Kibana is responsible for the visualization. Docker out of box supports some logging drivers such as JSON log driver and GELF log driver to handle the log streams of each container. With GELF log driver the container logs can be forwarded to an ELK stack. Graylog Extended Log Format (GELF) is understood by most of the log aggregating systems like Logstash or more obviously Graylog. Developers have to get as much information as possible to be able to take care of a trouble [Prakash et al. 2016]. Problems may cause automatic roll-back of the application to the previous stable version in a seamless way. The analysis of logs and monitoring data is application-specific and their evaluation may be difficult. Therefore, using big data analysis and machine learning shall be involved.

In this paper we argue for a new DevOps-style A/B testing for an automated, user experience-based approach. We take advantage of logging and monitoring features to get feedback from the end-users.

Our approach works in Docker containerized realm, thus the webapplications and every tool which are used in the evaluation run in containers. After the specified duration the A/B test is evaluated and winner version of webapplication remains in the production environment automatically.

This paper is organized as follows: in section 2 we present our A/B testing approach from high-level and go into implementation details per components in section 3. Finally, this paper concludes and presents future work in section 4.

2. OUR APPROACH

We propose an approach for A/B testing of webapplications in Docker containerized way. This approach takes advantage of Docker, Nginx server, ELK stack and GrayLog. We have developed a script for controlling the A/B testing. This script is written in Python.

The two variants of the same webapplication are running in separate set of containers. The Nginx server is also running in container. The Nginx is routing the users to A or B version based on their IP hash. On the client side of both webapplications HTTP requests are submitted to the Nginx server. Two kinds of requests are in-use. The first one is a periodical one which states if the user still using the application. The second one is triggered by the end-user to check the user's activity. Both requests contains the origin of the application version as tag. We collect the logs of webapplications in an ELK-stack.

The Python script runs on the host machine. The script takes a duration parameter that specifies how long the A/B test is running. When this duration expires the script gets log information from the ELK-stack container and evaluates which version is the better one. The script controls the Docker to discontinue the running of the worse version and replaces it with the better one.

3. TECHNICAL DETAILS

3.1 Client side

For our research we created two versions of a simple website with different title and headlines clearly indicating which version we are looking at using our web browser. Both version has a link.

The page also contains a JavaScript script which acts like a subset of any other webpage analytics bundle. It generates a UUID on every page load and sends a HTTP GET request to the '/ping' route sending the generated client id as parameter in every 5^{th} second. By these messages we can set a metric which can indicate how long our user stays on the page. We also send a HTTP GET request containing the client id on the click event of the link to the '/click' endpoint.

We do not create any relation between those UUIDs and session cookies to keep it anonymous like a good analytic tool anyway. We did not want to make any unnecessary (A or B) version specific code in the web page (nor the backend) because it could pollute the source code of the product itself and can be irrelevant of its aspects.

3.2 Backend side

The backend simply serves a static HTML file (which contains all of the client side code) and responds with status 200 on every request at route '/click' and '/ping'. It dumps every request to the standard output. All of these configurations have been done in a single `nginx.conf` file to keep this proof of concept project simple.

3.3 Docker containers

First of all, our testing stack has a load balancer container which listens on port 80 and forwards the requests to `node1` or `node2`. The forwarding depends on the client IP hash in order to make sure the

clients click and ping requests are being forwarded to the very same node which served the HTML file earlier (so the load balancer will not switch up the version between two requests from the same client).

The load balancer references Node1 and Node2 by their aliases. Docker Engine has a solution to create virtual networks between containers so when there are multiple products up and running containers on the same host machine they do not interfere with each other connecting to separate virtual networks. Docker compose takes care about creating a network for our project defined in a `docker-compose.yaml` file (see below) by default. This default network is created with the name of the containing folder (assuming it is the same as the project name) with a `_default` prefix. This came handy when we created a new container and connected it to the same network by hand on the and of the test evaluation.

The Docker Engine takes care about DNS services on the virtual network that is why we can reference containers by their names. We do not need to change configurations on every startup and we do not have to save IP addresses in environment variables or hosts files on containers. It is more dynamic and more secure.

```
error_log /dev/stdout info;

events {}

http {
    access_log /dev/stdout;
    upstream abtest {
        ip_hash;
        server node1;
        server node2;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://abtest;
        }
    }
}
```

Node1 only differs from Node2 in its `index.html` file and more importantly in its tag. Node1 has “version-a” tag while Node2 has “version-b” tag at the beginning. The version tag is also sent in every log message to the Graylog server providing the identity of the version. As shown at the code snippet below Node1 and Node2 have not got any open ports they can receive requests only through the load balancer.

As we mentioned earlier the backend prints all of its requests to the standard output. The standard output is forwarded in GELF format to the GELF server.

```
version: '3'

services:
    loadBalancer:
        image: nginx
```

```
    ports:
        - "80:80"
    volumes:
        - ./etc/nginx.conf:/etc/nginx/nginx.conf:ro

node1:
    image: nginx
    logging:
        driver: gelf
        options:
            gelf-address: "udp://127.0.0.1:12201"
        tag: "version-a"
    volumes:
        - ./nodes/static/versionA:/usr/share/nginx/html:ro
        - ./nodes/etc/nginx.conf:/etc/nginx/nginx.conf

node2:
    image: nginx
    logging:
        driver: gelf
        options:
            gelf-address: "udp://127.0.0.1:12201"
        tag: "version-b"
    volumes:
        - ./nodes/static/versionB:/usr/share/nginx/html:ro
        - ./nodes/etc/nginx.conf:/etc/nginx/nginx.conf
```

3.4 Log aggregation

There are numerous ELK stack configurations available on the Docker community hub so we omit the details for now. We have a Graylog server up and running which receives the logs of Node1 and Node2. We have set up an extractor which checks the message property of the log and uses regular expression to extract 'click' or 'ping' from the request route to a separate field called `clientLogEvent` when it is present and an other extractor works in the same way and extracts `clientSessionId`. Making extractors and testing queries on the Graylog web interface is comfortable and can be done without the need of digging in Elastic search querying. It is suitable for anyone who wants to shape it to fit their own specific A/B test scenario.

3.5 Evaluation and replacement

We have decided this task has to be done on a host machine by a script which can interact with the Docker Engine (or Swarm, Kubernetes, etc). For security reasons we cannot (neither want to) give a container access to other containers on system level.

We have chosen Python as the most suitable script language for this task. Python has maturity, and most of the *nix boxes have Python environment pre installed and another good reason is that Docker has a solid Python SDK, actively used by the Docker Compose project.

In our example we have decided to measure the count of click metric ('/click' route requests) – the bigger the better. When we exceed the duration of the test the script sends one query per version to

the Graylog server API to count its clicks (clientLogEvent: click). We use Apache Lucene syntax for queries. The script compares the result and then with the power of the Docker SDK shuts down the loser version Node and replaces it with an instance of the winner version.

The Python script itself interacts with the Graylog Web API and gets a session token by sending login credentials. At this point we could use API tokens set up on the Graylog Web interface, but we have not wanted to increase the complexity of the configuration for this example. The query is sent to the Graylog REST API, but it is just like any other REST API call so we omit the details for now. The interesting part is how we replace the container running the worse version with a new container running the better one. We stop and remove the "loser" container at first to avoid naming conflicts later on. After that we create a new container with the same parameters as the "winner" container, but with the name of the loser one. We connect the new container to the projects network using the same alias as the removed container had. When we start up the new container the load balancing works the same as before and the new node can be reached by the same name its predecessor could be reached by.

```
loser = client.containers.get(loserContainerName)
loser.stop()
loser.remove()

newNode = client.containers.create('nginx',
    name = loserContainerName,
    volumes_from = [winnerContainerName],
    log_config = {
        'driver': 'gelf',
        'options': {
            'gelf-address': 'udp://127.0.0.1:12201',
            'tag': winnerTag
        }
    }
)
bridgeNetwork = client.networks.get('bridge')
bridgeNetwork.disconnect(loserContainerName)

testNetWork = client.networks.get(self.networkName)
testNetWork.connect(
    loserContainerName,
    aliases = [loserContainerName]
)
newNode.start()
```

The script is just a proof of concept but we have created a command line interface for it because we have created some parameters so we can test it on different setups. Its help text tells us what parameters we can use for our test.

```
$ abtestCli -h
usage: abtestCli.py [-h] [--duration DURATION] [--aTag ATAG] [--bTag BTAG]
                    [--networkName NETWORKNAME] [--apiAddress APIADDRESS]
                    [--apiUser APIUSER] [--apiPass APIPASS]
                    aName bName
```

```
A CLI tool for runtest

positional arguments:
  aName
  bName

optional arguments:
  -h, --help            show this help message and exit
  --duration DURATION
  --aTag ATAG
  --bTag BTAG
  --networkName NETWORKNAME
  --apiAddress APIADDRESS
  --apiUser APIUSER
  --apiPass APIPASS
```

3.6 Running

Assume that we have the `docker-compose.yaml` file in our currently working directory.

```
$ docker-compose up -d
```

After it started up our services we only have to start our Python CLI script. It has three mandatory parameters:

(1) Duration – in ISO 8601 duration format
(2) A version container name
(3) B version container name

```
$ abtestCLI.py PT30M ab_node1_1 ab_node2_1
```

After thirty minutes the script will log the name of the better version and replace the worse with it.

4. CONCLUSION

A/B testing is a powerful method to improve software quality and user experience. It gains feedback from two akin versions of the same product (software, search ad, newsletter email, etc.) and it measures the end-user engagement.

We have developed an approach and related tools for executing A/B testing in Docker containerized environment. Our proof of concept implementation is working and has fulfilled our expectations but there is a lot of work to do and a numerous of choices to make before it becomes production ready. One of our goals was to keep the stack and the implementation simple to leverage the understanding of the conception.

We have mentioned that Docker Compose is for single host development and testing. And it did a great job providing us an initial state for our services. Also we have met the limitations of it such as dynamic configuration. Assume that we use the same stack, the A/B test is over and there are winner version containers everywhere then our system shuts down. Since Docker Compose cannot persist configuration changes to its compose file our configuration will be restored to the original one on the next `docker-compose up` command. There are great configuration management software tools like Puppet or Chef [Spinellis 2012]. Of course when it comes down to scalability we have to use Docker Swarm or Kubernetes client libraries, etc for managing version replacement on a multi-host system.

The concept is proven and we are excited to set it working on enterprise level. There could be a great A/B test deployment service on Amazon AWS or Microsoft Azure. Those companies have resources and technology to create a powerful analytics system with an integrated automatic deploy solution.

REFERENCES

David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (Sept 2014), 81–84. DOI:http://dx.doi.org/10.1109/MCC.2014.51

Daniel Cukier. 2013. DevOps Patterns to Scale Web Applications Using Cloud Services. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH '13)*. ACM, New York, NY, USA, 143–152. DOI:http://dx.doi.org/10.1145/2508075.2508432

Docker Inc. 2017. Docker Documentation. https://docs.docker.com/. (2017).

Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery* 18, 1 (2009), 140–181. DOI:http://dx.doi.org/10.1007/s10618-008-0114-1

Abdelkader Lahmadi and Frédéric Beck. 2015. Powering Monitoring Analytics with ELK stack. 9th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2015). (June 2015). https://hal.inria.fr/hal-01212015

Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. 2015. Dimensions of DevOps. In *Agile Processes in Software Engineering and Extreme Programming: 16th International Conference, XP 2015, Helsinki, Finland, May 25-29, 2015, Proceedings*, Casper Lassenius, Torgeir Dingsøyr, and Maria Paasivaara (Eds.). Springer International Publishing, Cham, 212–217. DOI:http://dx.doi.org/10.1007/978-3-319-18612-2_19

Tarun Prakash, Misha Kakkar, and Kritika Patel. 2016. Geo-identification of web users through logs using ELK stack. In *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*. 606–610. DOI:http://dx.doi.org/10.1109/CONFLUENCE.2016.7508191

James Roche. 2013. Adopting DevOps Practices in Quality Assurance. *Commun. ACM* 56, 11 (Nov. 2013), 38–43. DOI:http://dx.doi.org/10.1145/2524713.2524721

Andreas Schaefer, Marc Reichenbach, and Dietmar Fey. 2013. Continuous Integration and Automation for DevOps. In *IAENG Transactions on Engineering Technologies: Special Edition of the World Congress on Engineering and Computer Science 2011*, Kon Haeng Kim, Sio-Iong Ao, and B. Burghard Rieger (Eds.). Springer Netherlands, Dordrecht, 345–358. DOI:http://dx.doi.org/10.1007/978-94-007-4786-9_28

Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 275–287. DOI:http://dx.doi.org/10.1145/1272998.1273025

Diomidis Spinellis. 2012. Don't Install Software by Hand. *IEEE Software* 29, 4 (July 2012), 86–87. DOI:http://dx.doi.org/10.1109/MS.2012.85

Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. 2010. The Impact of Social Media on Software Engineering Practices and Tools. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, New York, NY, USA, 359–364. DOI:http://dx.doi.org/10.1145/1882362.1882435

Deepak Vohra. 2017. *Using an HA Master with OpenShift*. Apress, Berkeley, CA, 335–353. DOI:http://dx.doi.org/10.1007/978-1-4842-2598-1_15

Ya Xu, Nanyu Chen, Addrian Fernandez, Omar Sinno, and Anmol Bhasin. 2015. From Infrastructure to Culture: A/B Testing Challenges in Large Scale Social Networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*. ACM, New York, NY, USA, 2227–2236. DOI:http://dx.doi.org/10.1145/2783258.2788602