Continuous Benchmarking: Using System Benchmarking in Build Pipelines

Martin Grambow, Fabian Lehmann, David Bermbach
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
Berlin, Germany
{mg, flm, db}@mcc.tu-berlin.de

***Abstract*—Continuous integration and deployment are established paradigms in modern software engineering. Both intend to ensure the quality of software products and to automate the testing and release process. Today's state of the art, however, focuses on functional tests or small microbenchmarks such as single method performance while the overall quality of service (QoS) is ignored.**

In this paper, we propose to add a dedicated benchmarking step into the testing and release process which can be used to ensure that QoS goals are met and that new system releases are at least as "good" as the previous ones. For this purpose, we present a research prototype which automatically deploys the system release, runs one or more benchmarks, collects and analyzes results, and decides whether the release fulfills predefined QoS goals. We evaluate our approach by replaying two years of Apache Cassandra's commit history.

***Index Terms*—Benchmarking, Continuous Integration, Software Development, QoS, YCSB, Cassandra**

I. Introduction

Today's IT systems tend to be rather complex pieces of software so that even the smallest changes, either in the source code itself or in the application settings, can have a big (negative) impact on their performance, e. g., adding or configuring security features as shown in [1]–[3]. Besides increased latencies which result in a poor user experience, cloud based systems usually use autoscaling to automatically adapt the amount of resources to meet quality of service (QoS) goals. When a software change now leads to increased use of hardware resources, more resources will be provisioned leading to significantly higher cost. Finally, systems and services rarely operate in an isolated way. In interaction with other systems and services, however, small QoS changes will affect other services and may even start a butterfly effect in some scenarios. Regardless of the cause, these deficits are often coupled with reduced revenue or even fines if the current performance metrics do not meet the defined service level agreements (SLA) or user expectations, e.g., Google reports that the number of daily searches per user decreases if the latency of results increases [4].

In order to prevent undesired effects on performance or other quality metrics, we propose to add system benchmarking to the build pipeline of software systems. This way, developers can assert that a new release is at least as good as the previous release and that it complies with SLAs. In this regard, we make the following contributions:

1) We describe how QoS requirements can be integrated into the development process and how benchmarking can be used to enforce QoS goals.
2) We present a proof-of-concept prototype, including the corresponding Jenkins plug-in.

II. Background

In this section, we give a short overview of paradigms and systems used in this paper.

Version Control with Git: Git[1] is a very popular distributed version control tool. Among many other features, software developers can download the current version of the source code from a given repository (checkout or pull), commit changes to this software locally, and upload (push) them to ultimately merge them into the current version.

Continuous Integration & Deployment: Continuous Integration (CI) and Continuous Deployment (CD) are two modern paradigms which aim to improve, automate, and accelerate the software development process leading to shorter release cycles. CI defines the process of integrating new software changes into the master version, including adapting and running corresponding test cases which ensures that the software is extensively tested before it is merged into a production branch of a system [5]. CD describes the automated process of releasing and deploying new software versions. Once a new release has been thoroughly tested in the CI process, it is automatically rolled-out to the production system so that frequent daily releases are possible; this shortens the release cycle.

Both processes, CI and CD, are designed to run multiple times per day, depending on how many features are implemented per day and the release policy. Thus, 10 minutes are a guiding value for the total run time of both processes so that developers can get early feedback on their software changes. In practice, however, this is not always realistic so that multi-tiered deployment pipelines are usually used instead.

Jenkins: Jenkins[2] is an open source automation server for all tasks related to the software development process. Triggered by various kinds of events, a Jenkins server executes user-defined job pipelines which consist of multiple build steps;

[1]https://git-scm.com/
[2]https://jenkins.io/

structure and sequence of steps depends on the respective application. While the typical structure of a pipeline is a simple sequence of tasks, it is also possible to insert conditions or execute steps in parallel. Since different software projects have different needs, Jenkins can be extended through so called plug-ins which can be used to implement arbitrary functionality.

Benchmarking: Benchmarking "is the process of measuring quality and collecting information on system states" [6]. In contrast to monitoring, benchmarking aims to answer a specific question and is typically used to compare different system versions, configurations, alternatives, or deployments. Moreover, benchmarking typically measures the quality of a non-production environment with arbitrary metrics at a specific time in multiple test runs and analyzes its output in a subsequent offline analysis. Benchmarking can involve running micro benchmarks which measure very small isolated features down to single method performance. Typically, however, a system is deployed on several machines (along with other systems it potentially depends on) and a measurement client on another machine. Then, the measurement client runs an application-driven workload against the system under test (SUT) and tracks its changes in QoS.

Apache Cassandra: Apache Cassandra[3] is a popular NoSQL database system which was originally developed at Facebook. The system was explicitly designed for elastic scalability, high performance and availability; in exchange, it offers only eventually consistent guarantees based on the PACELC trade-offs [7].

YCSB: The Yahoo! Cloud Serving Benchmark[4] (YCSB) [8] is the de-facto standard benchmark for NoSQL databases. YCSB offers a suite of synthetic standard workloads which can be run against the SUT. After each execution, YCSB reports aggregated measurement results including throughput, latency, and the total runtime which can be used to evaluate and rank the compared systems or system configurations.

III. Approach

In this section, we describe how benchmarking can be used as part of a CI or CD build process to ensure QoS requirements, give an overview of our system architecture, and describe how builds with QoS problems can be detected in benchmarking results.

A. Continuous Benchmarking

As already described, state of the art CI and CD solutions focus on functional tests and micro benchmarks such as single method performance. To complement this, we propose to regularly run system benchmarks as part of the build process. Comparable to CI and CD, we refer to this as Continuous Benchmarking (CB).

CB should only be done if the correct functionality of the software has already been ensured, otherwise the benchmark might run against a buggy software version and produce incorrect results, e.g., because data records are processed much faster due to an error. CB should, hence, take place once all functional tests and integration tests have already been passed. In the following, we will give an overview of the steps involved in CB; see also figure 1 which gives a high level overview of the CB process and how it integrates into a CI/CD process.

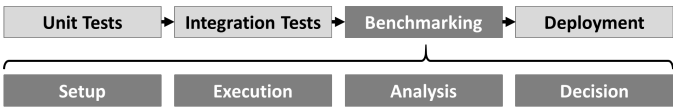

Fig. 1. Main Steps of Continuous Benchmarking

Setup: First, once all functional and integration tests have been passed, the SUT and the benchmarking client must be set up, which can be done either sequentially or in parallel. To get comparable results over several runs, it is important to deploy both systems in the same environment in each CB process (same hardware, operating system, supporting libraries, etc.). If, for example, a different hard disk would be used in each run, the different speed would have an influence on the results and it would not be possible to carry out a trend analysis of the key figures. Also, unless the benchmark explicitly targets a future use case, it is reasonable to choose a runtime environment as similar as possible to the production environment. Finally, the SUT and benchmarking system must be isolated from external factors which could affect the benchmark results, e.g., other processes running on the same machine, other services interacting with the SUT, or too much traffic on the network.

Depending on the system under test, it may also be necessary to deploy other external systems (e.g., BigTable [9] always relies on GFS [10] and Chubby [11]) or to run a preload phase which inserts an initial data set into the SUT [12].

Execution: In the second step, the benchmark is actually run. In fact, it may be run several times as benchmarks should usually be repeated and different benchmarks and benchmark configurations may be run in parallel. Here, monitoring should be used to assert that the machine(s) of the benchmarking client do(es) not become the performance bottleneck. Furthermore, benchmarks should be run for a sufficiently long time, typically, this means to keep a system benchmark running for at least 20-30 minutes [6].

Analysis: In the third step, the results from all benchmark runs need to be collected as they will typically be distributed across multiple machines. Next, these results need to be analyzed. Depending on the benchmark, this may mean simple unit conversions (e.g., ns to ms) and aggregations or more complex analysis steps. For instance, when data staleness is measured following the approach of [13], this may involve analysis of several GBs of raw text files.

Decision: Finally, the process needs to decide whether the current build is released to the deployment pipeline or whether the process is aborted because QoS goals were not met. Depending on the application and its benchmark, the measured values can either be compared with absolute thresholds, e.g.,

[3]http://cassandra.apache.org/

[4]https://github.com/brianfrankcooper/YCSB

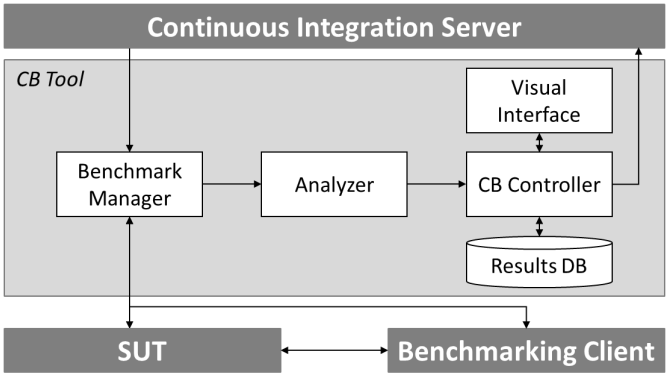

Fig. 2. Architecture and Main Components of Continuous Benchmarking Setups

as defined in Service Level Agreements (SLA) or software specifications, or relative thresholds could be used (we discuss these metrics section III-C).

B. Architecture and Components

As shown in figure 2, the components in our architecture are closely aligned with the steps described above. Typically, our CB process will be triggered by a CI server or some other build pipeline automation system. For this, the Benchmark Manager acts as the main entry point. Once it has been triggered, it installs and configures both the SUT and the Benchmarking Client before starting the execution of the benchmark run(s). Next, the Benchmark Manager collects all results and forwards them to the Analyzer which is responsible for all analysis steps. Of course, the Analyzer may also act as a proxy that forwards the raw results to an external analysis system – like the Benchmarking Client, the Analyzer is SUT-specific. Finally, the Analyzer forwards the aggregated analysis results to the CB Controller along with the raw data. The CB Controller then persists the data, decides on success or failure of the evaluated build, and reports the result back to the CI server. Beyond this, the Visual Interface visualizes the benchmarking results for human users and is also used to configure the CB Controller (e.g., to adjust relative and absolute thresholds).

C. Metrics

The final step of our approach requires some metrics to decide on the success or failure of a benchmark run. Depending on the system requirements, these decision can be based on fixed values given in SLAs, can reject a build because of a sudden and significant drop/jump in QoS compared to the last build, or detect a negative trend over multiple builds. Here, we present the decision algorithms which we will later use in our evaluation.

Fixed values (FV): The most simple method to detect undesired builds is to apply fixed thresholds, e.g., from SLAs. A build is rejected if the determined metric m_c, e.g. latency, is not in a specific interval.

$$f(m) = \begin{cases} succeed & \text{if } FV_{lower} < m_c < FV_{upper} \\ reject & \text{else} \end{cases} \quad (1)$$

Please, note that we always consider a lower and an upper value as thresholds. As an example, consider an application with a latency of 10ms. If this latency suddenly drops to 1ms, it could be the result of brilliant engineering. It is, however, much more likely to be the result of a bug where all requests terminate really quick with an error message.

Jump detection (JD): Especially if a software system offers much better quality than required by the FV thresholds, a sudden massive change in quality may still have a significant impact on the user experience. Thus, a relative comparison of the current run metric m_c to the predecessor run metric m_{c-1} can be used to reject builds in which QoS deviates from the last build by more than t percent. Concrete values for t obviously depend on the concrete application; however, we recommend values around 5%.

$$f(m_c, m_{c-1}) = \begin{cases} succeed & \text{if } t > 100 \left(\frac{m_c}{m_{c-1}} - 1 \right) \\ reject & \text{else} \end{cases} \quad (2)$$

Trend detection (TD): Finally, a longer lasting trend can lead to the software deteriorating slightly in b builds and finally exceed a given relative threshold of t percent in total. Here, the metric of current build m_c must not exceed t percent more than the moving average of the previous b builds (m_{c-b} refers to the metric of the bth build before the current one).

$$f(m, b) = \begin{cases} succeed & \text{if } t > 100 * \left(\frac{m_c \cdot b}{\sum_{i=1}^{b} m_{c-i}} - 1 \right) \\ reject & \text{else} \end{cases} \quad (3)$$

Besides these, there are other metrics which could be used. A valid approach might actually be to set the t value in JD and TD to zero to enforce continuously improving QoS. This, however, is likely to reject most builds unless the experiments are run on a completely isolated infrastructure.

IV. Evaluation

In this section, we evaluate our approach through a proof-of-concept prototype and a number of experiments with our proposed Continuous Benchmarking process in a realistic setup. We decided to use the existing commit history of Apache Cassandra as it is one of the most popular NoSQL systems and benchmark coverage, e.g., through the well established YCSB benchmark, is good. For our experiments, we replayed the commit history of Cassandra over the last two years.

A. Proof-of-Concept Implementation

We have implemented our system design as a proof-of-concept prototype. Parts of it are generic enough to be useful for all use cases, other parts are very use case-specific. Specifically, the Benchmark Manager's code depends to some degree on the SUT and the benchmarking client used. Here, we implemented everything as needed for our evaluation (see next section) with Apache Cassandra and YCSB.

The Benchmark Manager is implemented in Java and uses a number of Unix shell scripts for installation of Git, Ant, etc. if not already installed. For a production-ready implementation,

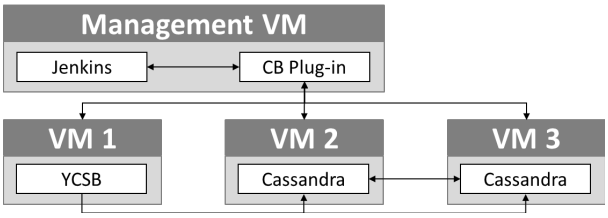

Fig. 3. Setup in all Experiment Runs

we would recommend to replace such shell scripts with "Infrastructure as Code" environments such as Ansible[5].

As already indicated above, the Analyzer is SUT- and benchmark-specific. In our evaluation case, it is implemented as a very short script that converts the YCSB output files to a standard format that our CB Controller can understand.

To better integrate our prototype into existing build pipelines, we have implemented the CB Controller and the Visual Interface as a Jenkins plug-in[6]. The Visual Interface allows users to specify thresholds and illustrates line charts for trend analysis as well as functionality for detailed insights into single benchmark runs. In contrast to our Benchmark Manager which is aligned with our experiments, the plug-in is generally applicable to arbitrary metrics.

B. Experiment Setup

For our experiments, we deployed Jenkins and our CB plug-in on a single virtual machine (VM). We configured the plug-in to run Cassandra on two other machines and YCSB on a third machine (see figure 3).

In all experiments, Cassandra used the "SimpleStrategy" for replication as we only had two nodes in the cluster; the replication factor was two. YCSB used workload A with the following configuration: fieldcount=10, fieldlength=100, records=20,000, operations=1,000,000 and threads=100.

We ran our set of experiments on Amazon EC2 m3.medium instances (3.75GB RAM, one CPU core) in the eu-west region, all in the same availability zone. We used the Amazon Linux AMI and ran all experiments on the same three VMs.

As input for our experiments, we used 465 commits between Jan 3, 2017 and Oct 23, 2018 of Cassandra's commit history which merged changes into the main trunk. We tested this reduced commit history three times successively, thus, each of these commits was benchmarked three times at different points in time, i.e., we had almost 1400 benchmark runs.

Please, note that a real build pipeline of course also involves steps such as testing. For our experiments, we decided to exclude these steps for reasons of simplicity.

C. Results

Figure 4 shows the results of our experiments as returned by YCSB. When ignoring outliers which can be expected when experimenting in the cloud [14], the values indicate the performance gradient as they are mostly densely packed.

[5]https://www.ansible.com/

[6]https://github.com/jenkinsci/benchmark-evaluator-plugin

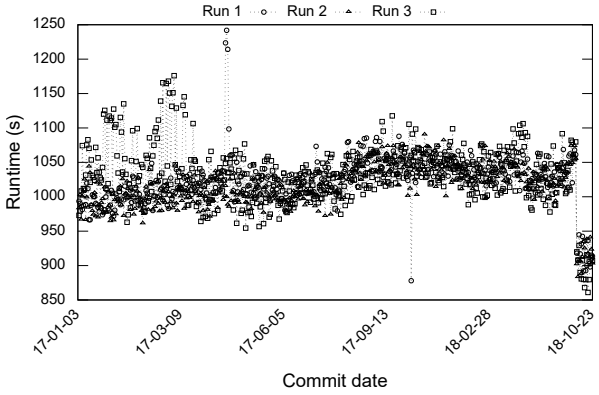

Fig. 4. Total Benchmark Runtime

At this stage, we did not specify any absolute or relative thresholds in our plug-in as we wanted the entire commit series to run through.

D. Application of Threshold Metrics

Following our approach and the metrics defined in III-C, we applied these thresholds on the median benchmark measurement to exclude outliers but still evaluate with actual measurement values. For the total benchmark runtime, we set $950s$ and $1100s$ as FV thresholds. We chose $t = 5\%$ as relative threshold for JD and $t = 4\%$ for TD which includes $b = 20$ builds.

Figure 5 illustrates these graphs along with the results from our median experiment run. An intersection of the median line and one of the other lines means that the respective build will be rejected.

The fixed value boundaries trigger only once – for the sudden performance improvement (ca. 13.5%) towards the end of the timeseries. Here, the developers introduced two features: "Flush netty client messages immediately by default" and "Improve TokenMetaData cache populating performance avoid long locking" which indicates that our detection is a false positive. Our jump detection algorithm would reject one build on May 10, 2017 (jump around 6.5%) which can either be caused by, according to the Git commit messages, "Forbid unsupported creation of SASI indexes over partition key column" or "Avoid reading static row twice from legacy sstables"; the first one, however, seems more likely to be the cause. The trend detection, on the other side, would reject 4 builds. The most significant violating build was on Aug 2, 2018 (performance drop of almost 4.6%) which just moves some code comments without touching any functionality, thus, the main reason for this negative trend is caused in the builds before and further analysis is necessary.

We only applied our defined metrics to the total runtime. Of course, there would be more metrics in other, more complex scenarios.

V. Discussion

Based on our measurement results, we believe that CB is a very useful approach for keeping QoS of a system either

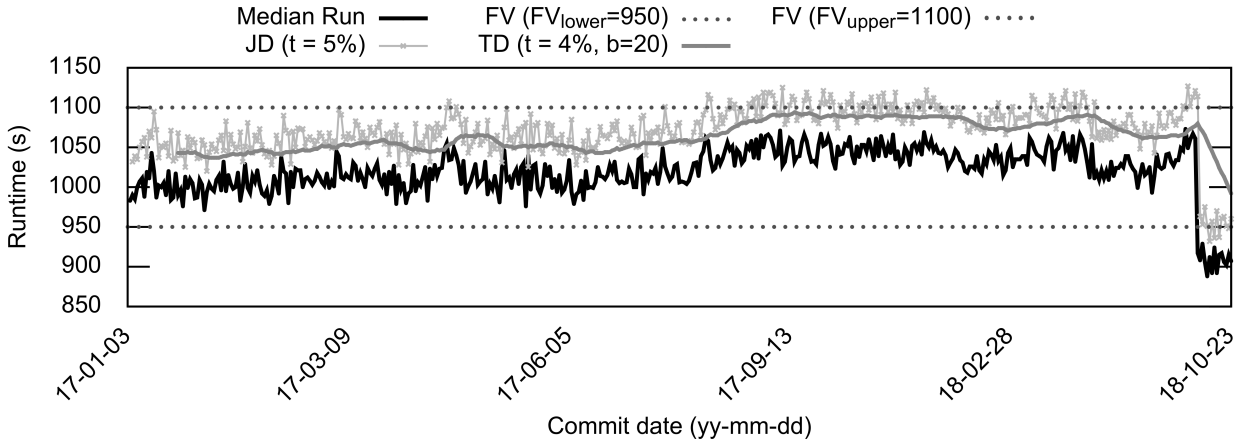

Fig. 5. Total Benchmark Runtime: Median Run and Threshold Metrics

constant or to continuously improve it while using the same amount of infrastructure resources. With our proof-of-concept prototype, we have also shown that the integration of CB into a build pipeline is indeed possible and does not involve a lot of effort – in fact, CB is simply integrated into the build process through our prototype which is automatically triggered for new versions. There is, however, also a number of open challenges and caveats.

Running CB will typically create additional costs for the development process; there is a tradeoff between how frequently CB is run, i.e., how early QoS problems can be detected, and the costs associated with that. We believe that this tradeoff is system-specific and cannot be solved in a general way. Developers also have to decide whether they plan to run CB on dedicated physical machines on-premises or whether they shift benchmark execution to the cloud. Depending on the frequency of CB runs, the on-premises option may be less expensive. The cloud option, in contrast, allows to run several benchmarks (and benchmark runs) in parallel so that it will be the preferred option when CB uses a set of benchmarks instead of a single benchmark only.

This choice between on-premises non-virtualized hardware and the cloud option is also related to the variance of results: We believe that running the CB process on dedicated hardware will produce more stable results with less variance across experiment runs. When running experiments in the cloud, we would recommend to run an initial experiment with at least ten runs (more is better) to get a better understanding of the variance effects caused by the underlying infrastructure. This would then also determine the number of necessary repetitions during the actual CB execution. Based on our AWS results, we would recommend to run the experiment at least three times (preferably five times) and to use the median result for further analysis and decision making.

There is also the question of when to trigger a CB run. In fact, we do not believe that running one for every single Git commit is the ideal but too expensive scenario. In our opinion, this would simply create too much data so that the developers may no longer be able to visually comprehend results. Comparable to our evaluation approach, we would recommend to trigger CB whenever a new feature branch is merged into the main branch. This also allows developers to manually override QoS thresholds when there are external events which mandate feature or configuration updates. For instance, when a new vulnerability in a TLS cipher suite is detected, switching cipher suites may be necessary but might have a strong impact on system performance [1]–[3].

There is also the challenge of finding a benchmark in the first place. In our case, we were running Apache Cassandra for which a number of open source benchmarks and benchmark tools exist. For some custom microservice, this will typically not be the case. In such scenarios, developers have to build their own benchmark first – comparable to test-driven development – which causes additional costs for personnel.

Finally, to conclude all cost aspects: CB will directly cause additional costs for the CB infrastructure and personnel costs for management or development of benchmarks. These costs, however, will likely be offset by indirect costs caused by unhappy customers or direct costs from compensation payments for SLA violations. Balancing these costs is a non-trivial task that is application-specific and should probably be approached in an agile way with continuous readaptation.

VI. Related Work

CB is a powerful mechanism for evaluating QoS of a new system version in a production-like environment. As such, it is relies on benchmarking approaches such as [8], [12], [15]–[19]. An alternative but also complementary approach to CB are live testing techniques such as canary releases [20] or dark launches [21]. In contrast to CB, live testing is characterized by the fact that a new version (of a software artifact) is directly deployed into the production environment in parallel with the older version.

For canary releases [20], this new version is initially rolled out for a very small subset of users and developers monitor its behavior in production. If there are errors or QoS issues in the new version, the impact only affects a few users and the version is reverted or shut down. Otherwise, more and more

users are added to the set of test users until the new version has completely been rolled out.

While canary releases aim to only affect a small subset of users in case of failures, dark (or shadow) launches [21], [22] eliminate potentially unsatisfied users completely by deploying a new version in the production environment without serving real user traffic – so called shadow instances. This way, no user is confronted with the new version and its potential issues.

Live testing techniques can be used to detect performance and other QoS issues in production. However, testing new versions in a production environment might be problematic for several reasons: First, a production system is usually in a normal state with usual load and regular traffic. Thus, a new version is never evaluated in production under extreme conditions or for rare corner cases. Second, a roll-out of several new versions of multiple software artifacts is administratively complex and error-prone, though tools like BiFrost [23] try to overcome these problems. Third, theoretical setups and architectures including new versions are hard to evaluate with live testing techniques. Finally, live testing does not necessarily create the right data to identify QoS degradation in the system release as varying workloads depending on user traffic will lead to varying observable QoS behavior. All this can be done with CB, e.g., by creating benchmark setups for extreme load peak situations. As benchmarking, however, can never be identical to a production load, we propose to combine the strengths of both approaches, i.e., to use both live testing and CB in parallel.

Comparable to our approach, Waller et al. [24] also proposed to include benchmarking in CI pipelines and present a Jenkins plug-in for this purpose. In contrast to our approach, however, they focus on measuring performance overheads of a code instrumentation tool. This is rather different from generic system benchmarking of distributed systems but supports the relevance of our approach.

Beyond these, there are two Jenkins plug-ins which could handle the task of our Visual Interface: the Performance[7] plug-in and the Benchmark plug-in[8]. Both, however, have explicitly been designed for small single-machine micro benchmarks such as single method benchmarks.

VII. Conclusion

Complex systems are very sensitive to change and even the smallest change can strongly affect QoS. Particularly, such changes occur frequently when releasing new software versions. Existing CI/CD pipelines, however, focus on functional testing or single method performance measurements and can, hence, not detect changes in QoS.

In this paper, we proposed a new approach called Continuous Benchmarking in which one or more system benchmarks are run as additional step in the build pipeline. Measurement results from these benchmark runs are then compared to either absolute thresholds, e.g., as specified in an SLA, or to relative thresholds which compare the result to previous results to assert that QoS levels always improve or at least remain constant across releases. We have prototypically implemented CB using Apache Cassandra as SUT and YCSB as benchmarking client and evaluated our approach by replaying almost two years of Cassandra's commit history.

[7]https://plugins.jenkins.io/performance

[8]https://github.com/jenkinsci/benchmark-plugin

References

[1] S. Müller, D. Bermbach, S. Tai, and F. Pallas, "Benchmarking the performance impact of transport layer security in cloud database systems," in *Proc. of IC2E*. IEEE, 2014.

[2] F. Pallas, J. Günther, and D. Bermbach, "Pick your choice in hbase: Security or performance," in *Big Data*. IEEE, 2016.

[3] F. Pallas, D. Bermbach, S. Müller, and S. Tai, "Evidence-based security configurations for cloud datastores," in *Proc. of SAC*. ACM, 2017.

[4] J. Brutlag, "Speed matters for google web search," 2009.

[5] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works*, vol. 122, 2006.

[6] D. Bermbach, E. Wittern, and S. Tai, *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[7] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *IEEE Computer*, vol. 45, no. 2, 2012.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of SOCC*. ACM, 2010.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proc. of OSDI*. USENIX Association, 2006.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proc. of SOSP*. ACM, 2003.

[11] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. of OSDI*. USENIX Association, 2006.

[12] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai, "BenchFoundry: A Benchmarking Framework for Cloud Storage Services," in *Proc. of ICSOC 2017*. Springer, 2017.

[13] D. Bermbach, "Benchmarking eventually consistent distributed storage systems," Ph.D. dissertation, Karlsruhe Institute of Technology, 2014.

[14] ——, "Quality of cloud services: Expect the unexpected," *IEEE Internet Computing*, 2017.

[15] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the weather tomorrow?: Towards a benchmark for the cloud," in *Proc. of DBTEST*. ACM, 2009.

[16] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proc. of VLDB Endowment*, vol. 7, no. 4, 2013.

[17] D. Bermbach and E. Wittern, "Benchmarking web api quality," in *Proc. of ICWE*. Springer, 2016.

[18] A. H. Borhani, P. Leitner, B. S. Lee, X. Li, and T. Hung, "Wpress: An application-driven performance benchmark for cloud-based virtual machines," *Proc. of EDOC*, 2014.

[19] D. Bermbach, J. Kuhlenkamp, A. Dey, S. Sakr, and R. Nambiar, "Towards an Extensible Middleware for Database Benchmarking," in *Proc. of TPCTC*. Springer, 2014.

[20] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[21] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, 2013.

[22] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proc. of SOSP*. ACM, 2015.

[23] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proc. of Middleware*. ACM, 2016.

[24] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including performance benchmarks into continuous integration to enable devops," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, 2015.