CanaryAdvisor: a Statistical-Based Tool for Canary Testing

Alexander Tarvo, Peter Sweeney, Nick Mitchell, Vadakkedathu Rajan, Matthew Arnold, Ioana Baldini
IBM Research
Yorktown Heights, NY, USA
atarvo,pfs,nickm,vtrajan,marnold,ioana@us.ibm.com

ABSTRACT

Canary testing is a novel technique that allows testing complex applications in a real-world environment. However, canary testing requires collecting and analyzing vast amounts of data coming from the testing system. Analyzing this data manually is a laborous task. In this paper we present CanaryAdvisor – a tool to automate canary testing of cloud-based programs. CanaryAdvisor continuously monitors the testing system and detects degradations in correctness, performance, and scalability in a new version of the program.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

Keywords

Software testing, performance

1. INTRODUCTION

Microservice architectures are redefining the way that software is designed and deployed as a suite of independently deployable services that communicate through a REST [1] (HTTP) interface. Traditional enterprise applications have been designed in three tiers: a front-end client-side user interface, which runs as javascript in a browser, a database, which stores data, and a server-side application, which handles the HTTP requests, interacts with the database and generates the content sent to the browser. The server-side application, which microservice architectures are targeting, are designed as a monolithic unit; that is, a single logical executable. That means that even a small change to the code base requires the whole monolithic unit to be rebuilt, tested and then redeployed, which happens infrequently because testing requires a large suite of tests to be run, which that execute for days and even weeks.

In contrast, microservice architectures breaks the server-side application up into a suite of loosely coupled microservices. Each microservice is independently deployed and scaled, and the loose coupling creates moduar boundaries. In this new world of a microservice architecture, deploying a new version of a microservices into a production environment has simplier testing requirement, because only the microservice functionality needs to be tested, but requires extra care at deployment to ensure that the entire service is not brought down or significantly crippled by the change in the microservice. Canary testing [1] addresses this problem by managing a microservice's deployment into a production environment. canary testing

During a canary test, a new version of an application (called the "canary") is deployed alongside the stable running version (called the "baseline"). Then a small portion of the user traffic is diverted to the canary. The behavior of the canary is compared to the behavior of the baseline. Any unexpected degradation in performance, correctness, or resource consumption by the canary would result in the test failure. In this case, all the traffic is re-routed to the baseline version of the application, and the canary is aborted (that is, the canary "dies"). Otherwise the canary is considered to be safe for larger-scale deployment, and replaces the baseline.

Canary testing significantly reduces the risk of introducing failures into the production version of the software. However, canary testing poses a unique set of challenges.

First, canary testing requires collecting metrics that provide complete and accurate picture of application's correctness and performance. Chosing informative metrics and collecting them without perturbing the system is nontrivial.

Second, the behavior and performance of a complex system in the cloud is non-deterministic. Virtualization, application co-location, and fluctuations in a workload cause variations in resource consumption, performance, or even reliability of the system. These variations, which are not related to the code change, should not influence the outcome of the test.

Third, manually determining if a canary should succeed or fail is difficult, because it requires monitoring a large number of metrics, detecting significant differences between baseline and canary, and interpreting the differences.

Fourth, failures and performance degradations must be detected as quickly as possible, so user requests could be rerouted back to the baseline version of the software. At the same time, the test results must be accurate. These are contradictory requirements, as making an accurate decision

[1]The name "canary testing" was inspired by using canary birds in the early days of coal mining to alert miners about the presence of toxic gases in the air.

ISSTA '15 Baltimore, Maryland

in the cloud environment may require significant amount of data and time.

Canary testing is actively discussed in the industry. However, it received little attention in academia. In this paper, we present a CanaryAdvisor – an automated tool for canary testing of software deployment. CanaryAdvisor works by continuously monitoring baseline and canary versions of an application. It collects values from system performance counters and application logs, and transforms them into metrics that characterize performance, correctness, and resource utilization of the system. CanaryAdvisor compares values of these metrics collected from the baseline and the canary and makes the decision on the outcome of this comparison.

To withstand short-term variations in metric values, CanaryAdvisor employs a statistical approach. The approach samples metric values to obtain a distribution, and then employs hypothesis testing to check if a parameter of the distribution such as a mean is statistically different between the baseline and canary. The information on the status test is displayed to the user in realtime through a web-based UI.

2. USER SCENARIO

To provide a necessary context, consdider a following example. Tanya is responsible for a microservice that comprises a business critical service that brings in a significant amount of revenue to her company. Tanya has made new functionality to her microservice that supports a new advertisement campaign. Initial unit and system integration tests have passed, and now Tanya wants to test the microservice with real workload. In particular, she wants to ensure that neiter 4XX and 5XX HTTP error codes, nor response time increase. So Tanya deploys the new version of the microservice on a Linux virtual machine in her datacenter, and passes information about this deployment to the CanaryAdvisor. Then she routes a portion of the traffic to the new version (canary).

The CanaryAdvisor continuously analyzes streams of data coming from the baseline and the canary versions of the micorservice. The input metrics has some amount of variance, so initially the CanaryAdvisor does not have enough information to ensure if any changes it observed are statistically significant. As the CanaryAdvisor receives more data, it detects that the percentage of errors in the HTTP log of the canary did not change compared to the baseline, and CPU utilization even decreased. However, the response time has high variance, so the CanaryAdvisor needs more data to make an informed decision. Thus Tanya lets the canary to run further.

After receiving more data the CanaryAdvisor detects a statistically significant increase in the response time of the canary version. Tanya immediately re-routes all the traffic back to the baseline version and uninstalls the canary.

After some investigation she founds a performance regression caused by inefficient use of locking. She fixes the performance regression and starts another canary test. This time the canary passes and becomes the new baseline by going into full production replacing the old baseline. Future canary tests will be against this new baseline.

3. TOOL ARCHITECTURE

This is a solution that solves Tanya's needs. From this example we see that canary testing is an open-ended activity, where test results must be analyzed and updated continuously. This naturally leads to the approach based on stream processing. Here CanaryAdvisor constantly processes streams of data from the applications and updates the decision. This involves the following stages: data collection, data pre-processing, metric comparison, decision making, and visualization. From the implementation standpoint, each stage is performed by the corresponding component of the CanaryAdvisor (see Figure 1).

Data collection. Data collection occurs directly on the computers that perform the test. The collection agents read the performance counters and application logs, do initial preprocessing (e.g. parsing HTTP logs), and send data to the next component of the testing tool. This approach is simple and lightweight, as it does not require instrumenting the software system under the test.

Data pre-processing. Raw data obtained with collection agents usually has an inappropriate format and thus are hard to analyze. During the data pre-processing stage data undergo series of transformations which convert it into the informative and actionable streams of metrics. We define the following metrics:

- $RT = rt_1, ..., rt_N$, where rt_i denotes the response time of i-th request. Here N is the total number of requests served;
- $ERR = err_1, ..., err_N$, where $err_i = 0$ if i-th request was served successfully and $err_i = 1$ otherwise.
- $SAT = sat_1, ..., sat_N$, where $sat_i = 1$ if $rt_i \leq T \wedge err_i = 0$ and $sat_i = 0$ otherwise. Represent user satisfaction over the request i. The request is considered to satisfy the user ($sat_i = 1$) if it was processed without error and with response time no more than T.
- $RC_{cpu}, RC_{disk}, RC_{net}$: represent the consumption of CPU, disk, and network by the application. Computed as the amount of resource, e.g. CPU time, consumed during the $[t; t + \Delta t]$ time interval and divided by the number of requests served during the $[t; t + \Delta t]$. This makes our metric invulnerable to changes in workload.

Metrics samples are annotated with the tag that denotes if they were obtained from the canary (e.g. RT^{can}) or from the baseline (e.g. RT^{base}) version of the program.

Metric comparison. During this stage CanaryAdvisor detects any significant difference between metrics received from the baseline and canary versions of the program.

To account for the inherent variance in the metric measurements we employ a statistical approach. We assume that each metric M follows some (unknown) distribution, which can be quantitatively described using some statistic $S(M)$. We use mean $\mu(RT)$ and $\mu(RC)$ as statistics for RT and RC metrics; proportion of erroneous requests $p(ERR) = (\sum err_i)/N$ and proportion of satisfied requests $p(SAT) = (\sum sat_i)/N$ as statistics for the ERR and SAT metrics.

Let M^{base} and M^{can} are versions of metric M obtained from baseline and canary versions of a program respectively. Now a significant difference $\Delta S(M) = |S(M^{base}) - S(M^{can})|$ between statistics $S(M^{base})$ and $S(M^{can})$ warns of a change in the behavior of the canary.

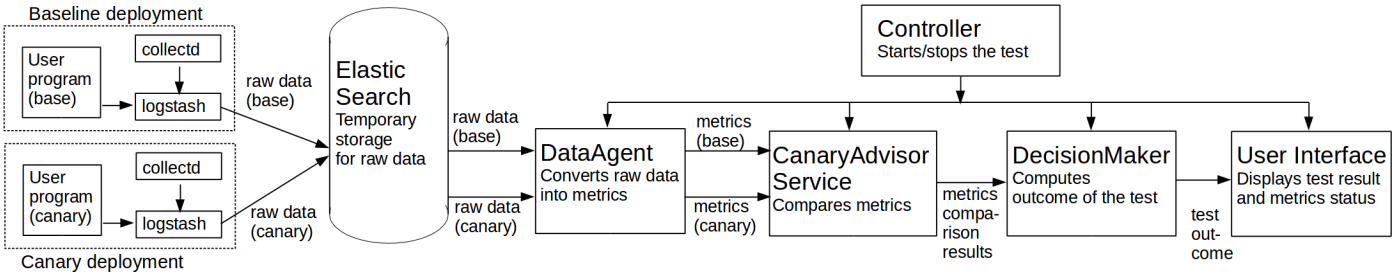

Figure 1: CanaryAdvisor architecture

However, true distributions for metrics M^{base} and M^{can} are not known. We have only samples of metrics collected from baseline and canary versions of the program. As a result, exact values of $S(M^{base})$, $S(M^{can})$, and, correspondingly, $\Delta S(M)$ cannot be computed.

Instead, the difference between means and proportions of two samples can be computed using a sampling distribution technique, as described in []. Namely, for significantly large samples ($N \geq 30$) the true difference $\Delta S(M)$ is located within a confidence interval $ci(\Delta S(M)) = (ci_{lo}, ci_{hi})$ with some pre-defined probability p. We claim that statistic $S(M^{can})$ has increased significantly compared to the baseline M^{base} if the lower bound ci_{lo} of that confidence interval is higher than 0 (namely $ci_{lo} > 0$). Correspodingly, $S(M^{can})$ has decreased if $ci_{hi} < 0$; $S(M^{can})$ didn't change significantly if $0 \in (ci_{lo}, ci_{hi})$.

To make this approach practical, we need to solve two problems. First, subtle differences in underlying hardware, OS, or presense of long-running tasks co-located with canary or baseline versions of the program may introduce a small bias in metric measurements. To ensure that our approach is resilient to such bias we introduce a tolerance factor x. Change in metrics is considered significant only if $ci_{lo} > x$ or $ci_{hi} < x$.

Second, in order to obtain accurate results for the test we must ensure that the width $W = ci_{hi} - ci_{lo}$ of the $ci(\Delta S(M))$ is small enough.

The width W represents the possible error in estimating the true difference $\Delta S(M)$. Unless $ci_{lo} > x$ or $ci_{hi} < -1 * x$, which indicate a significant difference between the canary and the baseline, we compare the width of the interval W to a pre-defined critical value W_{crit}. If $W > W_{crit}$ we attempt to decrease width W.

W is directly proportional to the sample variance and inversely proportional to the sample size N. The sample variance depends on factors that are beyond our control: effects of caching, fluctuations in the workload, presense of short-term tasks co-located with the program, etc. Thus the only way to decrease W is to collect more data before making any decision on metric M.

The algorithm for comparing canary to the baseline is outlined at the Figure 1

Decision making. During this stage CanaryAdvisor analyzes comparison results for all the metrics and decides on the outcome of the test.

Different metrics have different semantics and different importance. To allow flexibility in making a decision we define two boolean parameters for each metric M_j: *higher_is_better* and *important* .

Algorithm 1 Algorithm for comparing canary and baseline

```
if W > W_crit then
    if not ((ci_lo > x) or (ci_hi < −1 · x)) then
        more data required
    end if
end if
if ci_lo > x then
    S(M) has increased for the canary
else if ci_hi < −1 · x then
    S(M) has decreased for the canary
else
    S(M) remained the same
end if
```

If *higher_is_better = true* for M_j, then a decrease in $S(M_j)$ results in labeling M_j as being in a "worse" state. Similarly, the increase in $S(M_j)$ results in labeling M_j as being in a "better" state. *higher_is_better* is set to *true* for the SAT metric and $false$ for the remaining metrics.

Metrics with *important=true* are considered to be more important for the outcome of the test than remaining metrics. For example, ERR metric can be considered as an important one.

This leads to the following rules to determine the test outcome:

- Pass. All the metrics are in the "better" or "same" state;
- Fail. At least one important metric is in the "worse" state;
- Some problems. At least one non-important metric is in the "worse" state. No important metric is in the "worse" state;
- Inconclusive. Some metrics are in the "need more data" state. No metric is in the "worse" state.

Visualization CanaryAdvisor displays the current state of the test through a web-based UI. It displays the state of the test, and the state of each individual metric. In addition, for each metric M_j the UI displays the change in the statistics $S(M_j)$, as well the histogram of the metric values.

4. DISCUSSION AND LESSONS LEARNED

Discussion and lessons learned

5. RELATED WORK

Canary testing is gaining

6. CONCLUSION

conclusion

7. REFERENCES

[1] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

APPENDIX

This section contains a detailed set of demo steps and expected outputs of our tool.

A. DEMO STEPS

In this demo we will simulate three canary tests in a cloud environment. We will use a Daytrader application as our test subject. Daytrader is a Java application that simulates a stock trading platform. It offers a set of REST APIs for various actions, such as user logon, reviewing the portfolio, purchasing and selling securities.

Step 1: Start two versions of Daytrader on two Linux virtual machines via a shell script:

- "baseline": Daytrader instance with no changes in it. Imitates a baseline version of the application;
- "canary": Daytrader instance with an injected correctness and performance faults. Calling two of the program's REST APIs will results in generating an internal server error (HTTP 500 response code) with the probability of 20%. Response time for two of its REST APIs is increased by 30%;

Step 2: Start an instance of CanaryAdvisor (CA) using the Web interface. Open the UI of the CanaryAdvisor (see Figure 2 for an output example).

- Open the CanaryAdvisor UI page. Demonstrate that CA is in the "Inconclusive" state, as it lacks data to make decision about most of the metrics. Show the list of metrics grouped by the metric type: *RT*, *SAT*, and *ERR* metrics for each REST API entry of the Daytrader and *RC* metrics. Emphasize that CA also estimates the amount of time when the decision will be available;
- Query the output of the CanaryAdvisorService through its REST API. Demonstrate the list of metrics. Emphasize that although most of the metrics need more data to make a decision, the basic descriptive statistics, such as mean and variance, is available for each metric.

Step 3: Wait for 1-2 minutes. The CA will now enter the "Fail" state. It reports increased response time for some of the DayTrader REST APIs and increase in error percentage for other APIs.

- Click on the "Worse" button at the top. A list of metrcics in the "worse" state will be shown (see Figure 3);
- Click on one of the metrics. A window with a histogram for "baseline" and "canary" versions will appear. Note that the distribution of metric values is obviously different for canary and baseline versions.

Step 4: Stop the "canary" version of Daytrader using the shell script. Assume that discovered bugs were fixed. Re-deploy the "canary" version of Daytrader. Now this will be the version of the Daytrader without any changes, fully identical to the "baseline".

Step 5: Start an CA instance again.

- Open the CanaryAdvisor UI page again. Demonstrate that CA is again in the "Inconclusive" state. However, the time until decision is smaller than in the previous case;

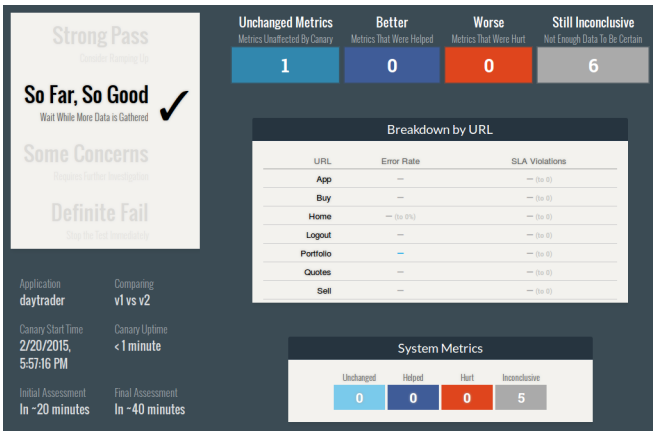

Figure 2: Not enough data to decide on test

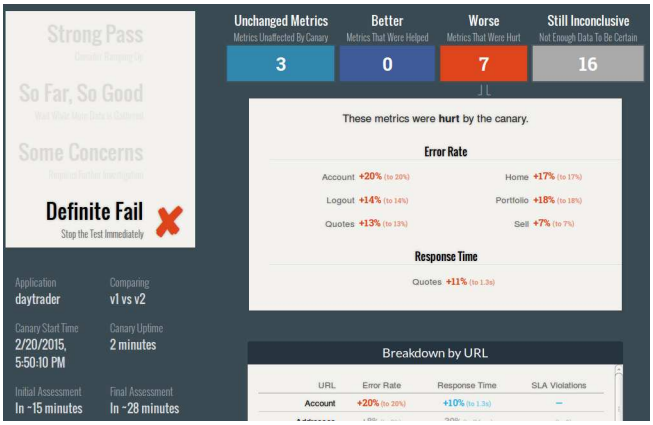

Figure 3: Canary test failed

- Query the output of the CanaryAdvisorService through its REST API. Demonstrate that for all the metrics, CanaryAdvisorService do not have much data about the canary version. However, it retrieved a large sample for the baseline version from the ElasticSearch database, which somewhat reduces time to decision.

Step 6: Wait for a few minutes. More and more metrics in the CA output will be transitioning from "Inconclusive" to "Unchanged" state.

- Click on one of the metrics in the "Unchanged" state. A window with a histogram for "baseline" and "canary" versions will appear. Note that the distributions of metric values is practically same for both these versions.
- Eventually the CA will enter into a "Pass" state (see Figure 4.

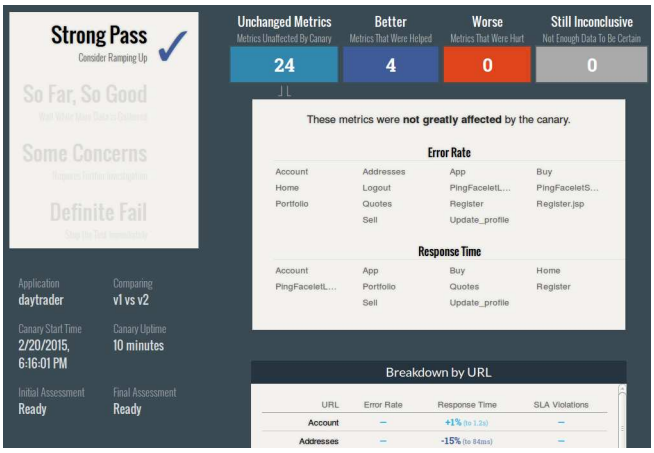

Figure 4: Canary test passed