Rapid Canary Assessment Through Proxying and Two-Stage Load Balancing

Dominik Ernst and Alexander Becker and Stefan Tai
Information Systems Engineering Research Group
TU Berlin
Berlin, Germany
Email: de,abe,tai@ise.tu-berlin.de

***Abstract*—Canary releasing is a means to check quality aspects of new software versions in a production environment, keeping risk to a minimum. We propose a novel approach to minimize the interference of canarying with the production system by (1) actively controlling request distribution in order to shorten the time a canary has to be running to achieve representative results and (2) unifying responsibility for that purpose and the collection of performance data about the canary on an ephemeral proxy load balancer. For (1) we contribute a two-stage load balancing approach, that implements weighted round-robin and a bucketing approach for HTTP-based services, that equalizes requests based on request URL. With explicit assumptions about infrastructure capabilities, we also describe the design of a proxy-based canary assessment tool to support (2). Both parts are implemented as a prototype and evaluated regarding feasibility, effectiveness and overhead. While the results cannot represent realistic production scenarios, they strongly indicate the validity of shortening a canary's lifetime by accepting a small overhead in client latency.**

***Index Terms*—software engineering, software testing, load balancing, continuous deployment, canary releasing**

I. Introduction

In modern software development approaches, continuous deployment (CD) helps to streamline releases of new versions and, by automating error-prone tasks, reduces the risk of each deployment. For the automation of build and test phases, multiple, well-established approaches and tools exist. Integration tests and, even more so, releases, are difficult to automate. Modern container orchestration and cluster management software simplify the management of complex deployments, but in turn increase complexity for setting up and maintaining cluster infrastructure. In practice, this discourages maintenance of a fully fledged test or staging environment. In this context, *canary releasing* [1] is becoming increasingly popular in combination with CD [2], because it follows the idea of deploying to the production system, while keeping risk to a minimum.

A canary is a small set or single instance of a new version of a software component. A portion of workload is routed to the canary and the canary's functional and non-functional behaviour is critically observed for a certain period of time. Deploying a canary into a running production system obviously causes the canary to be subject to variations, both caused by varying load and possible side-effects from other components. Furthermore, the integration of the canary constitutes a disruption to regular cluster management, because resources have to be provisioned or reconfigured for the time, during which the canary is active. The degree of changes to existing cluster configuration and the time for which those changes persist present two dimensions of interference, which canarying should aim to minimize, without compromising a valid assessment of the canary.

Our contribution in this area is a proxy-based approach, that aims at reducing the interference of canary releases on the production system. By utilizing an ephemeral proxy our approach minimizes changes to the configuration and infrastructure. At the same time, we propose an extension to load balancing strategies, to lower the required time to gather representative observations of an HTTP-based canary's performance, based on request paths. The evaluation of our prototype implementation for a two-stage load balancer shows an overhead in the low milliseconds, while reducing the difference in received load between a canary and reference deployment from 18.4% to 2.7% (for a total of 2000 requests with 100 different paths). Our prototype also comprises basic performance assessment and limits reconfiguration to a single component, showing the general feasibility of the proposed concept.

The remainder of this work is structured as follows. Section II introduces related work. Section III clarifies our assumptions and introduces terminology used throughout the paper. We contribute a comparison and reflection of architectural choices of integrating canary releasing with container orchestration in Section IV. We also discuss the angles of reciprocal influence between canaries and a production system there. Section V provides insight into our proposed proxy-based approach, from both design and procedural perspectives. Finally, in Section VI, we evaluate our prototype implementation. Section VII discusses limitations of our approach and elaborates on possible future work.

II. Related Work

Existing approaches from academia and industry for canary releasing focus on reduction of variability and facilitation of integration of a canary. However, there is a lack of structural clarity in research according to Rodriguez et al. [3], and little consideration of approaches to actively minimize interference

Pre-Print Version, ©2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/ICSA-C.2019.00028

of canarying. In fact, assumptions about the existence, capabilities and adoption of certain tools or infrastructure are commonplace and often not explicit.

Scherman et al., in [4], provide a model and prototype implementation for systematic live testing, called Bifrost. Their system is centred around a proxy, which is used for routing requests based on currently active testing methods. The authors include canary release as one mode of live testing and also evaluate overhead incurred by the Bifrost proxy. In contrast to the proposed approach in this work, the proxy is permanently running in front of all services that are subject to live testing and consequently also incurs an overhead while no canary-related adaptation of load balancing is active.

The authors of [5] present an approach and prototype, called CanaryAdvisor to evaluate a canary regarding its performance. In contrast to controlling the integration of the canary and managing overhead and variability this way, their approach focuses purely on the assessment part, i.e. supporting decisions on whether a canary version is "good". By employing statistical methods, such as normalizing load, the authors aim at differentiating and eliminating random noise from change-related deviations in performance metrics of the canary. Both Bifrost and CanaryAdvisor rely on external sources, such as monitoring systems, for performance measurements to support assessment of a canary.

Kayenta [6] is a platform for automated canary analysis and is integrated with Spinnaker [7]. Similar to CanaryAdvisor, it is intended to automate the verdict on whether a canary is good for release or not. Kayenta can be configured with different metric sources (e.g., time series databases, cloud providers' monitoring) and relies on the Mann-Whitney U test to asses significant deviations for specified metrics between canary and baseline. Spinnaker and Kayenta heavily rely on the existence of additional infrastructure, supporting required capabilities. More specifically, it requires not only a Kubernetes cluster, but also its own lifecycle management component, the adoption of compatible container build tooling, monitoring, deployment on one of three cloud providers and further configuration.

III. Canary Releases and Variability

Canarying is a way of releasing a new version of a software service into a production environment. For the context of this work, we assume canarying to happen in a microservices system or similarly complex, distributed system architecture, and to be done automatically in the context of CD. More specifically, we focus on canary releases where (a) services have dependencies and are never isolated, (b) are load balanced by either a dedicated load balancer or means of the infrastructure (within overlay networks, etc.) and (c) are horizontally scalable, i.e. multiple instances can co-exist and serve requests of clients in parallel. We assume that canarying is done in an unmodified production environment, i.e. no mirroring of requests to mocked downstream services is done. Finally, we make the assumption that requests are routed to service instances via both a global point of entry (for example a service registry or a gateway) and an individual (i.e., per-service) load balancer.

From an infrastructure perspective, we assume the existence of container and cluster orchestration, e.g., Kubernetes [8], Marathon [9] or others. Cluster orchestration is assumed to come with a few required capabilities:

- Ability to deploy software services as containers to hosts through an API.
- Functionality to horizontally manage scale of a deployment of services through an API.
- Basic load balancing for each service through a load balancer or proxy, with round-robin or random routing.
- A global point of entry to services, whose target endpoints for each service can be changed through an API

These requirements form a base, which is built upon for different architectural options, that are discussed in Section IV

From a procedural perspective, we assume a canary release to be a process consisting of four stages:

1) **Deployment** Canary versions and other required (infrastructure-) components are being deployed.
2) **Load Shifting** Load balancers and other supportive infrastructure are reconfigured to route a portion of requests to the canary.
3) **Observation** Monitoring, tracing and logging are used to collect both application-specific and generic metrics for the canary.
4) **Cleanup** Reversal of canary-related reconfiguration and un-deployment of components.

In practice, a canary release would be integrated with other steps of a continuous deployment pipeline, e.g., unit and integration tests before the canary release, an automated assessment based on key metrics collected during observation or a gradual rollout afterwards. A canary assessment requires capturing and comparing key metrics and a comparison of those metrics to previous versions. Good comparability of key metrics boils down to repeatability of the canary release process: if a repeated canarying of the *same* version produces significantly different results, a comparison of *different* versions is out of question.

An assessment of a canary within a CD context ideally would have the same desirable properties of a fully-fledged benchmark, that are: to be relevant, repeatable, fair, verifiable and economical [10]. However, in a production system, those criteria, and in particular repeatability, are difficult to achieve, because any meaningful running system will only have the exact same state and configuration for a very limited time. As such, canary releasing needs approaches to tackle both *request variability*, i.e., variations in current load caused by client load distribution, and *host variability*, caused by exogenous side-effects, e.g., from shared virtualized environments.

To handle host variability, existing tools and best practice approaches rely on a, so-called, *baseline* deployment. A baseline instance of a service is a newly created instance of the same version that is currently in production use and is created

in parallel and started simultaneously to the canary. Both baseline and canary receive the same portion of production workload and are deployed to identical environments. As a result they are comparable in their lifecycle, avoiding bias from performance deterioration over time.

Request variability is more difficult to eliminate, because it is directly related to client-induced load. It also depends on the type of a service (stateless vs. stateful) and a service's relative importance across the total number of client requests. As an example, a reproducible allocation of load for a stateless web service, receiving thousands of requests per minute, is easier to accomplish than for an order placement service, which receives two orders of magnitude less requests.

IV. Canary Integration

Canarying requires modifications to the existing production deployment of a service. However, depending on the approach to integrate and observe the canary, the degree of intrusiveness of those modifications will vary. Different modifications will come with different requirements regarding infrastructure and tooling. Based on the four stages as defined in Section III, we summarize different deployment-related options, describe implications for each of the phases, discuss external requirements and outline our proxy-based integration in comparison to other options. Figure 1 provides an overview of a deployment, following our proxy-based approach (called CanProx) to integrate a canary.

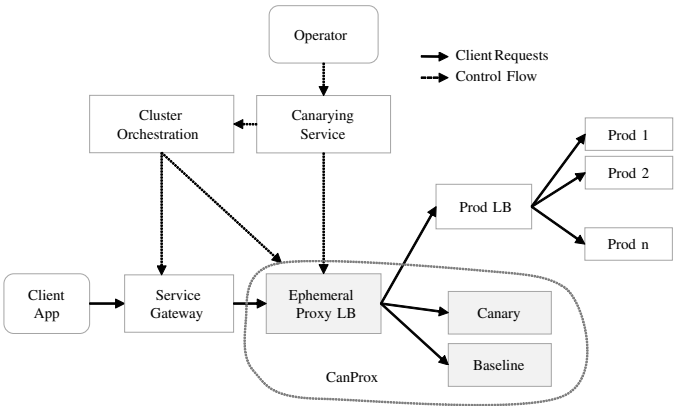

Fig. 1. Overview of a deployment including ephemeral proxy.

A. Canary and Baseline Deployment

As described in Section III, a *baseline instance* is a fresh deployment of the existing production version. How many baseline instances should be deployed and to which environments? Netflix report to typically deploy three canary and three baseline instances [11], but this obviously depends on the component that is being canaried. Host variability can be reduced by a deployment to a separate environment of both canary and baseline. We dub this *disjoint deployment*. However, with VMs, there is a chance that different VMs, even though identical in resources, exhibit different performance characteristics [12]. To mitigate this effect, co-deployment of one canary and one baseline instance to the same VM are an option, which we call *joint deployment*. In a production deployment built around container orchestration such as Kubernetes, services can be co-deployed with others and resource limits are available for containers. As a result, a single canary can be limited to remain well below 50% of a host's resources. There is no reported evidence that one option is better than the other, but using only one VM can shave off cost and reduce time to provision resources. It will also avoid bias towards downstream components by eliminating VM-placement sensitivities.

A third option is to deploy canary and baseline to any VM that currently has enough capacity. If resources in the cluster are very homogeneous and resource allocation is strictly enforced, this can be a cheap alternative. However, side-effects from the canary release on existing production load and vice versa have to be considered. For example, if a large number of measurements is taken during the observation phase, this could cause undesired congestion from monitoring and tracing. We call this approach *shared deployment*.

B. Load Balancing

After the deployment of canary and baseline, a portion of production requests are re-routed to them. Re-using an existing load balancer and injecting a new load balancer or proxy are the options we discuss in the following.

Re-using existing load balancing infrastructure requires the ability to re-configure said infrastructure at runtime. Many load balancers don't offer that capability. Second, the load balancer needs to support advanced load balancing strategies, if those are to be applied. Furthermore, a production load balancer may have many instances connected and the production infrastructure might be subject to autoscaling. As a result, high numbers and dynamically changing instances may affect overhead during a canary release, which also may impact production instances adversely. Re-configuring production load balancers, however, has the advantage of avoiding additional overhead from a second load balancer.

We propose *injecting of an ephemeral load balancer*, that takes the position the original (per-service) load balancer for the time of the performance assessment. Thus, canary releasing is possible in environments in which the original load balancer of the microservice does not support the required capabilities (weighted round-robin routing and runtime re-configuration). The main advantage of using an ephemeral load balancer is the ability to configure it specifically for canarying. A disadvantage is obviously the requirement of switching the endpoint to the newly created load balancer. The ephemeral load balancer is placed in front of canary and baseline, as well as the previously used production load balancer. Thus, effectively, two-stage load balancing is done for the time during which canarying is active.

C. Performance Measurement

Canarying can be used to discover both functional and non-functional issues resulting from changes to a software com-

ponent. We focus on non-functional properties, particularly performance. Performance, for web services, is captured by response time and throughput. Microservice deployments employ monitoring and distributed tracing to capture performance at runtime.

Existing tools for canary releasing assume the employment of these tools in order to record performance metrics and assess the canary afterwards. We dub those sources in the context of canarying as *runtime infrastructure*. To speed up canary assessment, monitoring and tracing can be increased in their level of detail (represented by log-levels and sampling, respectively). For example, instead of only capturing errors, warnings could be logged and instead of tracing every 1000^{th} request full tracing can be employed.

Instead of configuring canary, baseline and supporting infrastructure different from regular production deployment, an alternative is to rely on ephemeral components (such as the proxy) to capture application performance. We call those *canary infrastructure*. The proxy would implement full recording of request-response latency, avoiding changes to configuration of monitoring or tracing, which also eliminates possible side-effects in the assessment from those sources. By capturing performance at the proxy, however, effects of canary changes on downstream components can only be recognized, but remain opaque.

V. Proxy-based Canary Assessment

The proposed approach, specifically supporting a rapid and yet valid examination of a canary in the context of HTTP-based microservices, implements *joint deployment*, injects an *ephemeral load balancer* and relies on *canary infrastructure* for later assessment. Section V-A provides a design, followed by the canary integration procedure in Section V-B. Afterwards, we give insight into an equalizing load balancing strategy, (Section V-C), and finally provide further details of our prototype implementation of the aforementioned design.

A. Design

From a high-level perspective, the prototype is composed of two main components: coordinator and proxy load balancer (PLB), as shown in Fig. 2. The coordinator is responsible for executing all steps to conduct a canary release. Its principal activities include the provisioning of components, such as the canary and baseline instances, activation of interception of requests, and creation of reports. For that purpose, it takes input via its HTTP API (submodule *controller*), communicates with the a cluster management API (submodule *provisioning*) and collects and reports measurements from the balancer (submodule *reporting*). The PLB acts as a temporary load balancer in front of all instances of a component, receives requests and distributes them across production, canary and baseline instances of that component. It reports performance measurements about canary and baseline to the coordinator.

B. Procedure

For our prototype we map the introduced canary release process in Section III to a set of concrete steps. The primary inputs

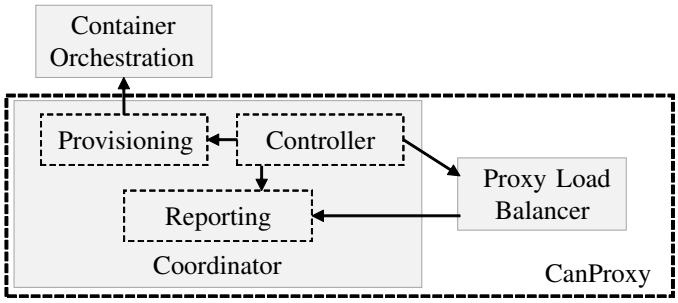

Fig. 2. Design of a proxy-based canary tool.

are (1) the original service reference, for which canarying is to be done, (2) the canary share and a reference to the container image to be used for the canary instance. The *deployment* phase consist of creating baseline and canary instances based on old and new images, respectively. Afterwards the PLB is spun up, configured with canary share and references to canary and baseline, as well as existing production instances. To that extent the coordinator interacts with the cluster management API to get references to and create required abstractions and containers. All created resources must be tracked by the coordinator.

For *load shifting*, the service reference (input (1)) is then simply pointed at the PLB and canarying is implicitly started. It should be noted that we assume a primary layer of load balancing or proxying, provided by the cluster management. In practice this would be an API gateway or, for example a Kubernetes Service resource. *Observation* is done by the PLB while it is active. After canarying is finished, the coordinator can be called to create a report of the canary release. A call to the coordinator's report API collects data collected by the PLB (e.g., also from an exposed API) and stores and/or presents them to a possible user.

Next, for *cleanup*, the coordinator once again relies on the cluster management API to reconfigure the service reference to point directly to production instances. Finally, the PLB, canary and baseline are un-deployed.

C. Load Equalizing Strategies

Same as for a benchmark, observation of a canary can only be considered relevant once a representative share of requests was received by the canary and baseline instances. In order to speed up canary assessment, creating an equal distribution of requests can help.

Round robin (RR)-based or randomized algorithms are used to create a fair allocation of load across all servers. An extension is weighted round robin (WRR), which traditionally is used to handle heterogeneous servers, e.g. regarding available resources. Any conditional load balancing algorithm increases the runtime complexity and by doing so, increases the latency of individual requests, because the load balancer has to keep and look up state. While in production settings, performance overhead of load balancing is critical, canarying always presents a disruption to regular execution. Keeping

the disruption minimal has two dimensions: duration and amplitude. By adopting a two-stage round-robin strategy, that considers both allocated weight of the canary and the URL of the called HTTP-based service, our approach aims at reducing duration while accepting a possibly higher request latency (amplitude). Which approach is to be favored in practice has to be decided by operators. For the proposed proxy-based

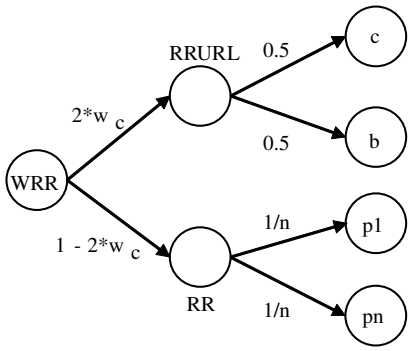

Fig. 3. Decision process of two-stage load balancing for canarying.

approach, we implement two-stage load balancing, as shown in Fig. 3. In the first stage, WRR, based on a weight for the canary w_c, a decision is made to route to production $p_{1..n}$ or canary c and baseline b. The second stage then applies a bucketing approach (RRURL for "round robin URL"), mapping calls to URLs to buckets, alternating routing between canary and baseline for each bucket. For the requests routed to production, a round-robin strategy is applied, indicated by RR. This could be any respective default strategy implemented for production instances, or requests could be forwarded to the previously used load balancer.

D. Prototype Implementation

To evaluate feasibility and effectiveness of our approach, we implement a prototype of the design described in Section V-A. For cluster management, we rely on Kubernetes, the proxy balancer is built around the NGINX Open Source [13] load balancer and the coordinator is implemented as a standalone component in JavaScript/node.js. PLB and coordinator expose HTTP APIs, with the latter providing a simple GUI to manually control the canary process as outlined in Section V-B. The implementation is to be understood as a proof-of-concept, our source-code is publicly available on GitHub[1].

VI. Evaluation

We evaluate our prototype regarding overhead (client-side request-response-latency) and compare the effectiveness of the WRRURL strategy in comparison to WRR. To that extent, we first describe our experimental setup and deployment in Section VI-A, followed by the descriptions and results of our experiments in Section VI-B. Section VI-C presents a short discussion of our findings. The scale of our experiments is not a realistic production one, but rather meant to reason about the feasibility of our approach and derive indications for which cases a practical adoption of our approach makes sense.

A. Setup

As a testbed, we use a Kubernetes cluster running on the Google Cloud Platform (GCP) using the hosted Google Kubernetes Engine (GKE). The cluster consists of five nodes hosting the components of our prototype (as described in Section V-A), a simple HTTP service and a load generator as shown in Table I. All components are deployed to the same region (europe-west1-b). Furthermore, we limit CPU resource consumption of all components significantly, values are given in Table I in brackets next to the components. Limits are at most half the available resources on each node to avoid throttling. It should be noted that Kubernetes system components occupy a small amount of resources on all nodes, but remain well below 0.5 vCPUs. Memory consumption and network bandwidth are never a bottleneck on any node, which was confirmed by running the experiments at five times the throughput without significant changes to the reported results. The *Service* is a simple HTTP-server implementation

TABLE I
Deployment of components on Kubernetes.

Node	vCPUs	Components
1	2	Coordinator (0.5 vCPU) and MongoDB
2	2	HTTP Service Instances (2 * 0.5 vCPU)
3	2	Canary & Baseline Instances (each 0.5 vCPU)
4	2	Proxy Load Balancer (0.5 vCPU)
5	4	Load Generator (2 vCPU)

written in node.js, that takes a parameter at startup, which adds a static delay before returning its response. It uses the Kubernetes Service abstraction as a default mechanism for load balancing, i.e. requests are routed in a randomized fashion. For our evaluation, we rely on artificial workload. For the load generator, we use a single instance of Vegeta [14] and generate n request lines in the format `GET http://test.service.domain/products/i` with $i \in [1, n]$. Vegeta uses these lines to produce requests in a round-robin fashion, the size n is varied for different experiments. All request lines produce the same HTTP response on behalf of the used web-service implementation.

B. Experiments and Results

We evaluate our prototype from a feasibility-perspective along two dimensions: First, additionally incurred *latency* caused by the PLB during canarying. Second, the *effectiveness* of the bucketing round-robin strategy RRURL is evaluated against RR. For both types of experiments, we use the same base parameters, which are summarized in Tab. II. The first two classes of experiments evaluate the overhead of our prototype. We configure a canary share of 10%, which is exactly mirrored by the baseline, so 20% of requests pass through stage two of the PLB as sketched in Fig. V-C. Furthermore, both baseline and canary are served by the identical implementation and configuration of the test webservice, as

[1]https://github.com/goodalex/canproxy

TABLE II
BENCHMARK PROPERTIES.

Property	Value
Response Body Size	75 bytes
Static Response Latency	10ms
Workload Generator CPU Cores	2
Worker Threads	8
Canary Share	10%

such there will (deliberately) be no difference in behaviour. For the first experiment, ten URLs are used (with uniform probability) and we compare the overhead of no canarying (1) against canarying with RR distribution (2) and canarying with RRURL distribution (3). The workload generator is set to 100 requests per second with a total runtime of 120s. Resulting latency measurements are shown in Fig. 4. Canarying increases the 99th quantile from 12.15ms to 13.97ms (RR) and 14.14ms (RRURL). Fig. 4 also indicates a slightly higher variance in request latency for the RRURL strategy over RR. Both in (2) and (3), WRR is used to distribute requests between production instances and canary/baseline. For the second set

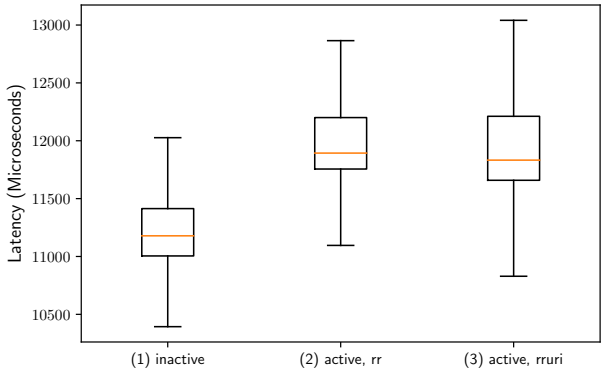

Fig. 4. Client-side latency (outliers not shown).

of experiments to evaluate overhead, we vary the number of URLs (10, 100 and 1000) and the canary share (1%, 10% and 50%) to compare effects of those parameters on the client-perceived latency. For both series, we additionally increase throughput to 200 requests/s and runtime to 300 seconds, in order to have more representative results for the borderline cases. All 99th quantiles for those experiments are shown in Tab. III. We find that both the number of URLs and the canary share have an insignificant impact on overhead, affecting only slightly the variance of results (not shown). The second dimension we evaluate is the *effectiveness* of the RRURL load-equalizing strategy over RR at the PLB. In order to do so, we compare the difference ratio between received calls per URL at canary and baseline. The PLB records detailed statistics of calls to each component per URL. The ratio is then computed as the sum of the difference in number of calls for each URL between canary and baseline, divided by the sum of requests sent to canary and baseline. We analyze the sensitivity of the

TABLE III
LATENCY MEASUREMENTS.

# URLs	99th Quantile
10	14.32ms
100	14.11ms
1000	14.14ms

(a) Overhead for # URLs

Canary Share	99th Quantile
0.01	13.74ms
0.1	14.32ms
0.5	14.06ms

(b) Overhead for Canary Share

approach depending on the number of URLs and total number of requests distributed. Request count is varied by runtime of the experiments, ranging from 50s to 500s, keeping throughput at 200 requests per second, while the numbers of URLs are five and one hundred to capture both extreme types of services. Results are shown in Fig. 5. As to be expected, a higher number of URLs increases the difference ratio significantly, especially for a low amount of requests.

C. Discussion

The presented results for the overhead dimension indicate a visible increase in client latency during canarying for both production and canary instances. We expected this increase due to the added complexity of load balancing algorithms employed at the PLB. The measured latency overhead of about 2ms during canarying indicates acceptability for many workloads. Further improvements of our prototype implementation may be possible. Using a different loadbalancer supporting those algorithms natively is likely to lower overhead, too. One shortcoming of our evaluation is the usage of the PLB as a singular "replacement" for any loadbalancing while the canary is active. In a more realistic setting, it would forward requests to production instances to the former load balancer. For our particular simplified deployment, however, this was not feasible as we rely on the Kubernetes Service abstraction as both our gateway and default per-service load balancer (c.f., Fig. 1).

Looking at the effectiveness of RRURL, we note that, for a high number of URLs, not even ten times the number of requests manages to achieve a similar ratio for RR than equalizing with RRURL. Inverting this viewpoint, we can assume that RRURL can shorten the time until a representative share of workload arrived at the canary and baseline by a factor of more than ten. Overall our experiments are done at very homogeneous load. With real-world workloads being much more skewed in the distribution of requests, e.g., exhibiting Zipfian or exponential distributions, we would expect an even bigger advantage of RRURL over RR.

VII. CONCLUSION

Canary releases replace and go beyond integration tests for complex, microservice-like architectures. Canarying is used to check non-functional properties of a component. In that regard, its goals are similar to benchmarking. But creating repeatable results is challenging, due to varying production load. Confidence in results of canarying can be increased by running the canary for a long time, or, which we set as our

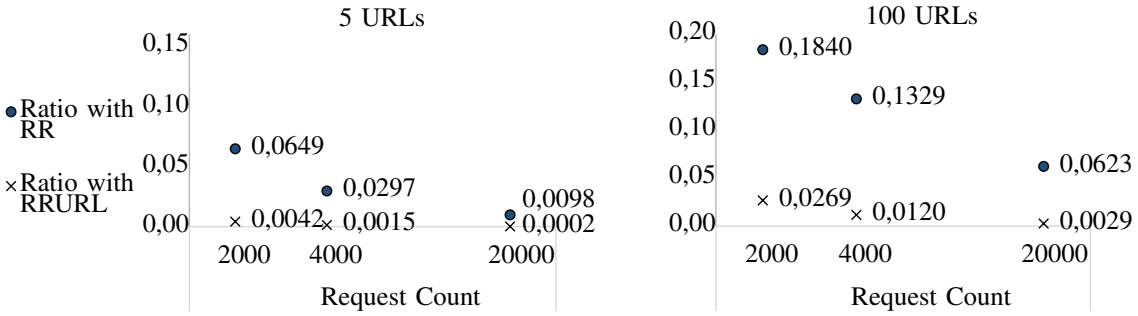

Fig. 5. Effectiveness of RRURL depending on total number of requests.

goal, actively creating a comparable load for canary and baseline. Another challenge in canarying comes from the required modifications to existing infrastructure. We propose to rely on an ephemeral proxy, that supports various requirements for canarying, including the aforementioned load balancing.

We implemented a prototype of a canary tool for HTTP-based services on Kubernetes. Our proxy load balancer, which comes with "batteries included" for load-equalizing algorithms and logging/monitoring, allows a significant reduction to the required re-configuration of production infrastructure to a single point. By implementing a two-stage load balancing algorithm based on request URLS (RRURL), comparability between canary and baseline can be sped up significantly. This comes at the cost of a short-term increase in client-perceived latency. We argue that, depending on the workload characteristics of canaried services, this may be favorable over running the canary over a longer period of time.

Our proof-of-concept and findings are limited to a HTTP-based microservice. We also don't regard possible interactions with other features of HTTP load balancers, such as sticky sessions, authentication or circuit breaker patterns. Nonetheless, the approach of bucketing requests by their type seems promising and likely can also be adopted for different protocols, for example RPC-based communication and within other components/layers. A future extension could be to apply statistical approaches to automatically subdivide requests into different classes before canarying. Both methods and classifiers would be interesting to compare.

While a proxy as the main point of capturing information about the canary simplifies canary assessment, re-configuring monitoring and tracing for canarying has its merits: measurements at the proxy obviously can only detect *if* there is an issue, but not what its cause is, especially if it occurs at downstream components. Propagating information about currently active canary deployments also will be sensible either way, for example in order to avoid false alarms, when monitoring thresholds are hit. However, we argue that monitoring and tracing are becoming part of a microservice infrastructure, which ideally should not be reconfigured at runtime ("Infrastructure as Code").

Design, implementation and optimization of methods applied in the context of canarying present themselves as opportunities for future research. More detailed examination of trade-offs for specific types of workloads are lacking, for example on the correlation between the duration of a canary release and the likelihood to obtain representative results. Statistical approaches to evaluate the validity of observations collected by canarying also present themselves as opportunities for further automation and optimization.

References

[1] D. Sato, "Canary release," https://martinfowler.com/bliki/CanaryRelease.html, 2014, last accessed: 2018-10-31.

[2] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[3] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, pp. 263–291, 2017.

[4] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:14. [Online]. Available: http://doi.acm.org/10.1145/2988336.2988348

[5] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini, "CanaryAdvisor: a statistical-based tool for canary testing (demo)," *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pp. 418–422, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2771783.2784770

[6] "Kayenta: Automated Canary Service," https://github.com/spinnaker/kayenta, last accessed: 2018-11-30.

[7] "Spinnaker - Continuous Delivery for Enterprise," https://www.spinnaker.io/, last accessed: 2018-11-30.

[8] "Kubernetes: Production-Grade Container Orchestration," https://kubernetes.io/, last accessed: 2018-11-30.

[9] "Marathon: A container orchestration platform for Mesos and DC/OS," https://mesosphere.github.io/marathon/, last accessed: 2018-11-30.

[10] K. Huppler, "The art of building a good benchmark," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5895 LNCS, pp. 18–30, 2009.

[11] "Automated Canary Analysis at Netflix with Kayenta," https://medium.com/netflix-techblog/automated-canary-analysis-at-netflix-with-kayenta-3260bc7acc69, last accessed: 2018-11-30.

[12] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920902

[13] "NGINX," https://www.nginx.com/, last accessed: 2018-11-30.

[14] "Vegeta: HTTP load testing tool and library," https://github.com/tsenart/vegeta, last accessed: 2018-11-30.