Topology-Aware Continuous Experimentation in Microservice-Based Applications

Gerald Schermann[1(✉)], Fábio Oliveira[2], Erik Wittern[3], and Philipp Leitner[4]

[1] Software Evolution and Architecture Lab, University of Zurich, Zurich, Switzerland
schermann@ifi.uzh.ch
[2] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
[3] IBM, Hybrid Cloud Integration, Hamburg, Germany
[4] Chalmers | University of Gothenburg, Gothenburg, Sweden

Abstract. Continuous experiments, including practices such as canary releases or A/B testing, test new functionality on a small fraction of the user base in production environments. Monitoring data collected on different versions of a service is essential for decision-making on whether to continue or abort experiments. Existing approaches for decision-making rely on service-level metrics in isolation, ignoring that new functionality might introduce changes affecting other services or the overall application's health state. Keeping track of these changes in applications comprising dozens or hundreds of services is challenging. We propose a holistic approach implemented as a research prototype to identify, visualize, and rank topological changes from distributed tracing data. We devise three ranking heuristics assessing how the changes impact the experiment's outcome and the application's health state. An evaluation on two case study scenarios shows that a hybrid heuristic based on structural analysis and a simple root-cause examination outperforms other heuristics in terms of ranking quality.

1 Introduction

The ever-increasing need for rapidly delivering code changes to fix problems, satisfy new requirements, and ultimately survive in a highly-competitive, software-driven market has been fueling the adoption of *DevOps* practices [2] by many companies. DevOps promotes the *continuous deployment* [13] of code to production, breaking the traditional barrier between development and operations teams and establishing a set of software development methodologies heavily based on tools to automate software builds, tests, configuration, and deployment. To further increase development agility, companies are frequently following a *microservice-based* [10] software architecture style. Microservice-based architectures are an evolution of the idea of service-oriented architectures [5,20], in which applications comprise a multitude of distributed services.

The agility facilitated by DevOps practices and microservice-based architectures enables companies to perform *continuous experiments* [16], which test the

E. Kafeza et al. (Eds.): ICSOC 2020, LNCS 12571, pp. 19–35, 2020.
https://doi.org/10.1007/978-3-030-65310-1_2

functionality and performance of new versions of application components under production load. A common embodiment of continuous experimentation is to perform *canary releases* [6]. In this practice, which resembles testing in production, one compares the test version (the "canary") of a microservice against the current version (the baseline) with respect to performance and correctness. Initially, the canary is exposed to requests of a small portion of users. If its performance and correctness remains acceptable, it is gradually exposed to more users until it replaces the baseline. If it fails to perform as expected at any time, all traffic is shifted to the baseline and the canary is terminated. Crucially, determining the health of a canary requires (1) collecting and storing the metrics of interest, and (2) comparatively analyzing the baseline and canary metrics.

Previous work [3,18] on assessing the outcome of continuous experiments considers the microservice under test in isolation, focusing on service-level metrics alone. These approaches ignore the fundamental principle that microservices communicate with each other and that these interactions affect the overall application behavior. For example, performance issues in a canary version of a service propagate delays (e.g., higher response times) within the network and when solely judging on isolated service-level metrics, multiple services could appear to misbehave. Given the scale of modern microservice-based applications compounded by a myriad of possible inter-service dependency patterns, identifying the root cause of such issues is challenging, especially when multiple microservices are under experimentation, e.g., running multiple canaries simultaneously.

We contend that continuous experimentation in microservice-based applications must consider the topology underlying all inter-service calls so as to allow developers to evaluate new versions holistically as opposed to in isolation. Out of dozens or even hundreds of identified (topological) changes it is crucial to assess those in detail that cause effects on the application's health state. Therefore, we propose an approach to not only identify and visualize changes between baseline and canary versions, but also heuristics to rank these changes based on their potential impact with the ultimate goal to guide developers when assessing continuous experiments. We implemented our approach as a research prototype that supports analyses in the context of multiple experiments running in parallel. Our approach starts with inferring *interaction graphs* for both the baseline and canary versions from distributed traces collected from microservice-based applications. We then compare these interaction graphs to identify topological changes, and rank these changes. A visual frontend allows developers to review specific changes and associated quality metrics (e.g., response times).

In summary, this paper makes the following contributions: (1) a characterization of topological changes that occur in microservice-based applications; (2) a general approach for ranking those observed changes; (3) three concrete ranking heuristics as embodiments of this approach; (4) a proof-of-concept implementation; and (5) an evaluation of the quality of the produced rankings.

Our evaluation shows that a heuristic combining principles of both structural analysis and performance analysis performs best across our evaluation scenarios.

2 Related Work

Previous research has empirically assessed continuous experimentation practices and challenges [15,16]. These works analyze reports on continuous experimentation practices by selected companies [8,17], and also present data collected more broadly using interviews and surveys. They find that software architectures based on components that can be deployed and operated independently (e.g., microservices) are essential for continuous experimentation, but also attest that root-cause analysis of observed problems is challenging. Our work attempts to address these challenges by considering the interactions in which updated services participate.

Multiple methods and systems have been proposed for continuous experimentation. *Kraken* is a system proposed by Facebook [19] for traffic routing between services, servers, or even data centers to identify performance bottlenecks using actual user traffic. *Bifrost* [14] formalizes continuous experiments consisting of multiple phases. Experiments that are specified in a domain-specific language are automatically executed by a middleware using smart traffic routing. The *MACI* framework [4] for management, scalable execution, and interactive analysis presents an alternative way to express experiments integrating recurring tasks around experiment documentation and management, scaling, and data analysis with the goal of reducing specification efforts.

The work by Sambasivan et al. [11] is the closest to our approach. It compares distributed traces to diagnose performance changes, distinguishing between *structural* changes and ones in *response-time*. While Sambasivan et al. assume similar workloads for the variants, our approach focuses on the topology and on experimentation settings to assign only a small fraction of users to experimental variants. Due to our set of change types, the comparison between the experimentation variants is more fine-grained in our approach. This does also apply for comparing our approach with Kiali[1], a tool that helps observing services within service meshs such as Istio[2]. While Kiali provides some basic health assessment, our approach dives deeper by not only analyzing topological differences but also ranking them to guide developers assessing the overall application's health state.

Ates et al. [1] proposed *Pythia*, a framework making use of distributed tracing to automatically enable instrumentation such as logs or performance counters on those layers (e.g., application, operating system) that are needed to diagnose performance problems. Santana et al. [12] investigates how syscall monitoring in combination with a proxying approach can be used to obtain and inject tracing-related meta-information with the goal to avoid code changes in the application to propagate trace information. Our work relies on distributed traces collected by the Istio service mesh using Envoy[3] proxies in combination with Zipkin[4] to infer topologies of microservice-based applications.

[1] https://kiali.io/.
[2] https://istio.io/.
[3] https://www.envoyproxy.io/.
[4] https://zipkin.io/.

3 Characterizing Change Types

In the following, we characterize recurring change types we identified when comparing service topologies. For this purpose, we derive formal representations of microservice-based applications and service-interaction graphs that frame our basis to define topological change types.

3.1 Microservice-Based Application

A microservice-based application $\mathcal{A}$ consists of a set of interacting services $\mathcal{A} = \{s_1, s_2, \ldots, s_n\}$. Services are available in different versions, e.g., stable version 1 of the *frontend* service and a new experimental canary version 2 depicted in Fig. 1 (Left). For a service $s_i \in \mathcal{A}$ this is represented as a tuple $\mathcal{VS}_i = \langle s_{i,1}, s_{i,2}, \ldots, s_{i,m} \rangle$, where $s_{i,1} \ldots s_{i,m}$ are the corresponding versions j of service s_i with $1 \leq j \leq m$. Note that Fig. 1 (Left) not only represents our running example, but also depicts a topological difference which we will cover in detail in later sections when we revisit this example.

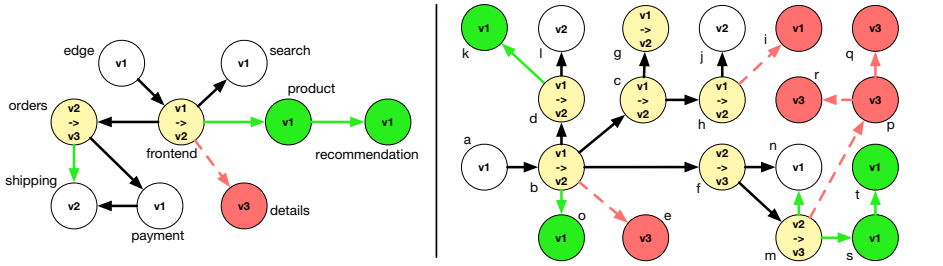

Fig. 1. Topological difference graphs of microservice-based sample applications. Left: running example (scenario 1). Right: scenario 2. Green depicts added functionality or calls, red depicts removed functionality or calls, and yellow depicts service version updates. (Color figure online)

In the context of continuous experiments a microservice-based application is available in multiple *variants* $\mathcal{VA} = \langle va_1, \ldots, va_p \rangle$ at the same time. An application variant comprises a combination of services $\langle s_i, \ldots, s_k \rangle$ with $i \leq j \leq k$ and $s_j \in \mathcal{A}$. For each of those services $s_j \in \mathcal{A}$ a concrete version u with $s_{j,u} \in \mathcal{VS}_j$ is selected. In Fig. 1, the *baseline* variant of the application includes version 1 of *frontend*, while the *canary* variant includes the new version 2 of *frontend*.

3.2 Interaction Graph

In a microservice-based application, version j of a service s_i interacts with other services by calling one or more of their endpoints. In our model, this interaction is represented by a directed graph $G = \langle V, E \rangle$ in which V and E denote sets of vertices and edges respectively. Every service $s_{i,j}$ of an application corresponds

to a vertex $v \in V$ in the graph, referring to version j of $s_i \in \mathcal{A}$, where $s_{i,j} \in \mathcal{VS}_i$. A directed edge $e = s_{i,j} \rightarrow s_{u,v}$, where $e \in E$, represents a call from a service $s_{i,j}$ (subsequently named *caller*) to another service $s_{u,v}$ (*callee*).

3.3 Topological Change Types

The presented formal model allows us to construct interaction graphs for every application variant and to compare them. Comparing interaction graphs of two or more variants reveals changes at the topological level. For example, in Fig. 1, when the canary version 2 of *frontend* is deployed, we observe that a new service (*product*) is required while the *details* service is no longer called.

In the following, we characterize typical change types that surface in the evolution of microservice-based applications. When comparing interaction graphs G_1 and G_2, every such change type appears as a certain pattern involving a subset of the vertices. We distinguish two categories of change types: *fundamental* and *composed*, where a *composed* change type is a combination of multiple *fundamental* change types.

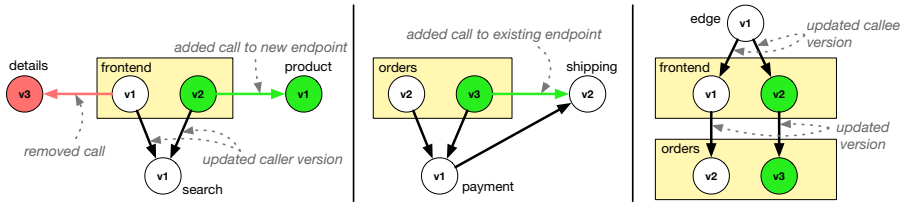

Fig. 2. Topological change types demonstrated on sample application (excerpt). Left: add call to new service, removed call, and updated caller version. Center: add call to existing endpoint. Right: updated callee version and updated version.

Fundamental Change Types. Fundamental change types involve calling newly added services (or service endpoints), calling endpoints of existing services, or removing calls to service endpoints.

Calling a New Endpoint. This change type represents new functionality manifesting as a call to a new resource, such as a service or a service endpoint that was added. In both interaction graphs G_1 and G_2 there exists a vertex (or node) representing a service a, but in different service versions: i in case of G_1 (i.e., $s_{a,i}$), and j in case of G_2 (i.e., $s_{a,j}$). The interaction graph G_2 contains an edge $e \in E$ with $e = s_{a,j} \rightarrow s_{u,v}$ calling a service u in version v that does not exist in graph G_1. Figure 2 (left) depicts this change type in our running example. The *frontend* service of the *canary* variant (version 2) calls a newly added *product* service that does not exist in the *baseline* variant (version 1).

Calling an Existing Endpoint. This change type characterizes reusing functionality, i.e., a new call to an existing service endpoint is made. There are again

two nodes in the interaction graphs representing the same service a, but in different service versions: $s_{a,i}$ in G_1 and $s_{a,j}$ in G_2. Graph G_2 contains an edge $e \in E$ with $e = s_{a,j} \rightarrow s_{u,v}$ denoting a call to service u that also exists in graph G_1; thus, $s_{u,v}$ is represented by a vertex $v \in V$ of G_1. However, there is no direct interaction (no edge) between $s_{a,i}$ and $s_{u,v}$ in G_1. Figure 2 (center) shows this change type in which the *canary* variant of *orders* (version 3) calls *shipping*. The *shipping* service is also part of the *baseline* variant involving version 2 of *orders*, but there is no direct interaction between *orders* and *shipping*.

Removing a Service Call. This change type represents the inverse of the previous one. A previously used resource is no longer used. Revisiting the previous change type, this time the interaction graph G_1 contains an edge $e \in E$ with $e = s_{a,i} \rightarrow s_{u,v}$ representing a call to a service u, but no equivalent edge between $s_{a,j}$ and $s_{u,v}$ exists in G_2. However, the service u might still be used in G_2 by other services. Figure 2 (left) represents this change type between the *canary* variant of *frontend* (version 2) which no longer calls *details*.

Composed Change Types. These change types are constructed from fundamental change types and denote updated caller version, updated callee version, and updated version.

Updated Caller Version. When comparing interaction graphs G_1 and G_2, the version of a calling service a is "updated". This caller-side version update is a combination of *removing a service call* and *calling an existing endpoint* change types. From the perspective of G_2, the service $s_{a,i}$ no longer calls a service endpoint $s_{u,v}$ (i.e., removed service call), but the same service a of the updated service version $(i \rightarrow j)$ is adding a call to $s_{u,v}$ (i.e., calling an existing service endpoint). Figure 2 (left) depicts an example. In the canary, the *frontend* service is updated to version 2, and both version 1 and version 2 call the *search* service.

Updated Callee Version. This change type represents the case of a version change in the service that is called. This callee-side version update combines *removing a service call* and *calling a new endpoint* change types. From the perspective of G_2, the service $s_{a,i}$ no longer calls a service $s_{u,v}$ (i.e., removed service call), but the same service $s_{a,i}$ calls a new version x of service u (update: $v \rightarrow x$, i.e., calling a new endpoint), hence there exists an edge $e = s_{a,i} \rightarrow s_{u,x}$. Figure 2 (right) exemplifies this change type when the version of *frontend* that is called by *edge* is updated from version 1 (*baseline*) to version 2 (*canary*).

Updated Version. This change type is a combination of *updated caller version* and *updated callee version* change types. There exists a service a and service u in both interaction graphs G_1 and G_2. In G_1, there is an edge $e_1 = s_{a,i} \rightarrow s_{u,v}$, and in G_2, there is an edge $e_2 = s_{a,j} \rightarrow s_{u,x}$. Hence, in G_1 the interaction happens between versions i and v of the services a and u, and in G_2 between versions j and x. From the perspective of G_2, both the caller and the callee versions are updated. Figure 2 (right) shows this pattern between *frontend* and *orders*. While for the *baseline*, version 1 of *frontend* calls version 2 of *orders*, in the *canary*, version 2 of *frontend* requires version 3 of *orders*.

4 Ranking Identified Changes

This section covers (1) the construction of the graph-based topological differences, (2) a generic algorithm that traverses these differences to produce a ranking of identified changes, and (3) three embodiments of this algorithm in the form of heuristics to assess the impact of the changes identified.

4.1 Constructing the Topological Difference

Our approach relies on distributed traces of a microservice-based application to (1) infer interaction graphs for each variant of the experiment and to (2) construct a graph-based topological difference resulting from their comparison.

Inferring Interaction Graphs. Distributed tracing is a technique used to collect information about calls between microservices. A *trace* is a set of data about the sequence of all inter-service calls resulting from a top-level action performed by an end user. Each call is associated with timestamped events corresponding to sending the request, receiving the request, sending the response, and receiving the response. In our approach, a developer needs to specify the application variants of interest, i.e., versions of services for baseline and canary and the experiment start time. Given the inputs, we then divide collected distributed traces of baseline and canary variants into clusters, where each cluster contains multiple interaction graphs (as defined in Sect. 3) with the same *root request*. A *root request* is a service call made to an edge service of the application, which in turn triggers other inter-service calls within the application, forming an interaction graph. In each cluster we also compute statistics on metrics for each inter-service call, namely, duration, timeouts, retries, and errors.

Comparing Interaction Graphs. The next step is to compare corresponding baseline and canary clusters of interaction graphs to identify topological changes based on the types described in Sect. 3.3. Once the changes and their types are identified, the graphs are merged into a single graph forming an "extended" topological difference (e.g., Fig. 1). The topological difference contains all the changes identified, their assigned type, and further statistics that were captured during the interaction graph's construction. Due to the merge, the difference graph contains also those structures (services and their interactions) that are common to the graphs under comparison. Doing so preserves the "big picture" and enables detailed analyses on the entire service network.

4.2 Traversing the Topological Difference

Once the graph-based topological difference is built, we execute a two-phase graph-traversal algorithm, consisting of the *annotation* and the *extraction* phases.

Basic Algorithm. In a first step, all vertices (or nodes) in the graph without *outbound* calls are visited (and marked as such). Then, the algorithm visits those vertices calling service endpoints that have been flagged as visited, marking them as visited again. This process is repeated until all nodes in the graph are visited.

Annotation Phase. In our approach, every node in the graph-based topological difference has an associated state $\mathcal{T}$, which is used to store any information to reason about, and ultimately rank changes. In the *annotation* phase, these states are set to hold information required for the concrete implementation of the ranking algorithm (i.e., heuristic). During a node's visit, a wide range of information is available, including the involved endpoint, outgoing calls and their change types, statistics (for either one or for both variants) that were computed during the construction of the interaction graphs, and any other queryable monitoring information (e.g., from Prometheus[5]). It depends on the concrete implementation of a heuristic which information is used and how it is combined.

Extraction Phase. In this phase, every node is revisited with the goal to *extract* a score $\mathcal{S}$ for each interaction (i.e., outgoing edge). Due to the nature of our change types, an interaction in the topological difference graph could comprise two edges in the source interaction graphs. The scoring happens on the change type level: edges belonging to the same change are merged. Edges that are common (without any change) in both source interaction graphs are treated as a special change type. The idea of the extraction phase is to rely on the state information gained in the annotation phase and to transform it into scalar values. Formally, this scoring function has the type signature $score : change \rightarrow int$.

Ranking. Once scores for all edges in the difference graph are computed, the scores are sorted in descending order and ranks from 1 to k are assigned, where k is the number of edges in the graph-based topological difference. The edge achieving the highest score is ranked on position 1. Equal scores leading to tied ranks are possible, even though they appear rarely.

In the following we will cover three specific embodiments of our algorithm. Starting with the *Subtree Complexity* heuristic, followed by the *Response Time Analysis* heuristic, we will cover their joint variant, the *Hybrid* heuristic.

4.3 Subtree Complexity Heuristic

This heuristic analyzes sub-structures of a topological difference and considers uncertainty in the context of experiments.

Concept. The graph structure is broken down into multiple subtrees (see Fig. 3 for an example). The fundamental idea of this heuristic is that the more complex the structure of the (sub-)tree is, the more likely it contains changes that affect the outcome of the experiment and the application's health state.

Initially, every node a has an assigned state of $\mathcal{T}_a = 0$. Whenever a node a is visited during the algorithm's annotation phase, its state $\mathcal{T}_a$ is set to $\mathcal{T}_a = \sum_1^n \mathcal{T}_i + p_{a,i}$ being $1 \leq i \leq n$ the (child) nodes of the outgoing calls of a. Thus, the state values $\mathcal{T}_i$ of called nodes i are summed up and weights $p_{a,i}$ representing individual *propagation* factors for these calls are added. During the *extraction* phase, for every interaction of a node a with a node i, the score for this edge e is computed as follows: $\mathcal{S}_e = \mathcal{T}_i + c_{a,i}$. Thus, the score is built from the state value $\mathcal{T}_i$ of the node (i.e., service) that is being called and an individual *scoring* factor $c_{a,i}$ for the edge.

[5] https://prometheus.io/.

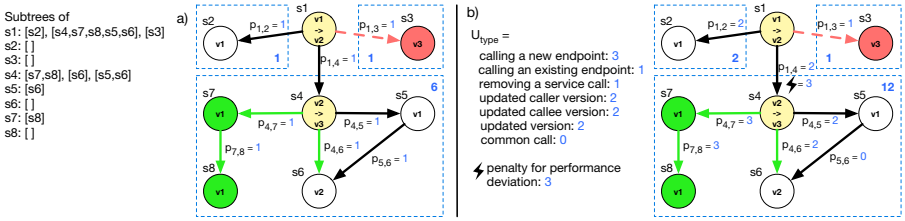

Fig. 3. Example of (topmost) subtrees in a topological difference. **a)** Basic subtree complexity (ST) in blue (i.e., counting the number of edges in a subtree). Service $s1$ has three subtrees. The state value of $s4$ is 5 (3 subtrees, 5 edges in total). Thus, the extracted score for the edge between $s1$ and $s4$ is $5 + 1 = 6$. **b)** Extended subtree (ST Ext) in blue, propagation values $p_{a,i}$ based on U_{type} values assigned to change types. Extracted score for the edge between $s1$ and $s4$ is $10 + 2 + 3 = 15$. (3 represents the performance penalty). (Color figure online)

The distinction between *propagation* and *scoring* factors serve the following purposes. The propagation factor directly influences the state values of the nodes (and thus the individual scores) when walking up the tree. This is useful if severe issues within a subtree are detected that should be reflected in the ranking of the changes. The scoring factor only influences individual scores, e.g., a single change. It allows expressing fine-grained differences among the changes. Depending on how propagation and scoring factors are chosen, the subtree complexity heuristic allows for multiple variations. Within the scope of this paper, we focus on two variations: *Subtree* and *Subtree Extended*.

Subtree (ST). This standard variant of the heuristic analyzes the structural complexity of the difference graph by counting the number of edges within subtrees. Propagation and scoring factors $p_{a,i}$ and $c_{a,i}$ are set to 1 for all edges independent of their change types. Figure 3a depicts an example in blue.

Extended (ST Ext). This variation introduces the concept of *uncertainty*. Calling entirely new services compared to calling a new version of an existing service leads to a different degree of uncertainty when assessing the application's health state. For the former, no information to compare to (i.e., previous calls or historical metrics) exists, while for the latter calls to the new version can be compared with previous calls. Deviations in metrics, such as response times or error rates, can be considered. Similarly, when a new call to an existing endpoint is made, even though a direct comparison on the interaction-level is not possible, there are still metrics available that are associated to the called service allowing an assessment whether this added call introduces unwanted effects. In our approach, we built upon these subtle differences in uncertainty for the identified change types and assign a weight U_{type} to each of them.

For the *extended subtree* heuristic, instead of the number of edges, the uncertainty values U_{type} associated to the individual edges' change types are summed up within a subtree. Hence, individual propagation factors $p_{a,i} = U_{type}$ are set to the uncertainty value of the edge's change type. Figure 3b depicts an example.

The rationale for this is to emphasize the uncertainty of subtrees involving many changes. Scoring factors are defined as $c_{a,i} = U_{type} + P$. Similar to the propagation factors we use the uncertainty values U_{type} and we introduce penalties P that are added to those interactions for which deviations are measured, e.g., significant changes in response times. This mechanism allows us to account for performance issues without running in depth root-cause analyses. Penalization applies to all interactions for which direct comparisons between the variants on the edge-level are possible, i.e., composed change types and common calls.

4.4 Response Time Analysis Heuristic

This heuristic tries to identify services and changes that have caused performance issues by incorporating the notion of uncertainty.

Concept. The intuition here is that in case of performance deviations (e.g., response time) spotted at a node, the node's surrounding changes that add additional calls (e.g., *calling a new endpoint*, or *calling an existing endpoint*) are potential sources of these deviations. This heuristic focuses on the overall response time (i.e., how long did the called endpoint take to respond) extracted from tracing data. However, the concept can be extended to incorporate other metrics that have similar cascading effects. Further, note that these performance comparisons are only possible for specific change types, namely composed change types and common calls.

The state $\mathcal{T}_a$ of a node a is extended to keep track of deviations and their potential sources while traversing the graph. It involves *flag*, a counter that keeps track how often a node is considered as the source of a deviation, a map *deviations* that stores which outgoing call (i.e., key) causes how much deviation (i.e., value, in milliseconds), and a list *source* keeping track which child caused the deviation. Algorithm 1 illustrates the analysis executed for every outgoing call in the *annotation* phase when visiting a node a.

Algorithm 1: Response Time Analysis

```
Input: node, child, call
if call.hasDeviation() :
    node.state.addSource(child)
    if len(child.state.deviations) == 0 :
        node.state.addDeviation(call=call,deviation=call.deviation)
        child.state.flag := 1
    else:
        flagSources(child)
        total := sum(child.state.deviations)
        node.state.addDeviation(call=call, deviation=max(call.deviation, total))
        if call.deviation > total :
            inc(child.state.flag)
            for c in child.calls :
                if c.type in [call_new_endpoint, call_existing_endpoint] :
                    inc(c.target.state.flag)
                    child.state.addSource(c.target)
```

In case of a deviation, the called child is added as a source. If there are no stored deviations for the child node, then the deviation is added to the node's state, and the child's state *flag* counter is set to 1. If there are deviations, the

recursive function *flagSources* walks through all the stored sources that might caused the deviation on the child's side and increases their *flag* counters. In the next step, the sum of all stored deviations (i.e., *total*) is calculated and the deviation is added to the node's state. If the call's deviation is higher than the total sum of deviations on the child's side, then it is likely that a change introduced this new deviation. Therefore, the child's *flag* counter is increased and the child's surrounding changes are analyzed. This involves all of the child's outgoing edges with *calling a new endpoint* and *calling an existing endpoint* change types. The target nodes of these edges are added as potential sources and their flag counters are increased.

By using different *scoring* factors in the heuristic's *extraction* phase we distinguish two variations: RTA and RTA Ext. The *annotation* phase (i.e., flagging) described in Algorithm 1 is the same for both variations.

Response Time Analysis (RTA). In the *extraction* phase, for every outgoing call of a node a to a child node i, the score for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_{i.flag}$. The resulting score corresponds to the final value of the child node's *flag*. Consequently, those services with the highest flag counts are ranked first.

Extended (RTA Ext). For this variation we revisit the concept of uncertainty and reuse weights U_{type} as scoring factors. Again, the rationale is that those interactions with high uncertainty for a change should have higher scores. To have a mechanism to balance between *flag* and *uncertainty* values, we introduce a *penalty* constant C. The scoring function for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_{i.flag} * C + U_{type}$.

4.5 Hybrid Heuristic

More complex (sub-)structures are more likely to contain changes that could cause problems. This is the strength of the subtree complexity heuristic. However, in case of performance deviations, the response time analysis heuristic provides more detailed analyses to identify the origin of problems. The goal of the hybrid heuristic is to combine the strengths of both, structural and performance analyses. The underlying mechanics of both heuristics remain untouched for the hybrid heuristic. During the algorithm's annotation phase, both the structural and the performance analyses are conducted. The extraction phase shapes how the individual results of the heuristics are transformed into a single result. We distinguish two variants: **Hybrid (HYB)** and **Extended (HYB Ext)**.

Both variants use the *extended subtree* heuristic (ST Ext) to determine state values $\mathcal{T}_i$. To determine state *flag* values, the standard variant of the heuristic uses *standard RTA*, while the extend hybrid variant uses *extended RTA*. Consequently, the scoring function for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_i + U_{type} + \mathcal{T}_{i.flag} * C$, being C the penalty constant established in RTA Ext, which is set to 1 in the case of the standard hybrid variant.

5 Ranking Quality Evaluation

To demonstrate our (formal) approach we developed a research prototype with the goal to assist developers on experiment health assessment and decision-making. The paper's online appendix[6] provides screenshots of the user interface (also depicting those two scenarios), source code of the heuristics, and a comprehensive replication package.

We evaluated the quality of the produced rankings on two concrete scenarios: (1) revisiting the running example, and (2) dealing with multiple breaking changes. Before we dive into details of the ranking quality evaluation, we briefly describe our evaluation's setup.

5.1 Setup

The setup involves a description of the method we used to assess the quality of the produced rankings, how we calibrated the parameters the heuristics are operating on, and how we generated the distributed tracing data.

Method. Normalized discounted cumulative gain (nDCG) [7] is a measure of ranking quality, widely used in information retrieval. Based on a graded *relevance* scale of documents in the result list of search-engine queries, DCG (or its normalized variant nDCG) assesses the usefulness (i.e., the gain) of a document based on its position in the result list. The gain of each document is summed up from top to bottom in the ranking, having the gain of each result discounted the lower the rank, which has the consequence that highly relevant documents ranked at lower positions are penalized. The DCG accumulated at a particular rank position p is defined as $DCG_p = \sum_{i=1}^{p}(rel_i/log_2(i+1))$.

rel_i is the relevance of the document at position i. Instead of documents we rank identified changes. In order to use DCG, the authors assessed the relevance of every single change of our two scenarios. In total, including sub-scenarios, 6 relevance assessments were conducted rating changes on a scale from 0 (not relevant) to 4 (highly relevant). We use a normalized DCG (nDCG) producing relative values on the interval 0.0 to 1.0, this allows for result comparison across scenarios. 1.0 is the maximum value representing a ranking with the most relevant changes on the top positions. As tied ranks are possible (e.g., changes with the same score and rank as resulting from a heuristic), we applied the nDCG adaption proposed by McSherry and Najork [9] considering average gains at tied positions.

Calibration. To calibrate the heuristics we followed an iterative exploratory parameter optimization procedure across all scenarios. For nDCG we considered the top $3, 5, 7$, and 10 positions of the ranking to be compared. For the penalties P and C used in the heuristics' scoring functions we iterated through values $1, 3, 5, 7$, and 10. We tested four different mappings of *uncertainty* values to change types U_{type}. Based on more than 9000 calibration results, we determined that

[6] https://github.com/sealuzh/topology-experimentation-appendix.

$P = C = 3$ and an uncertainty mapping U_{type} (i.e., *change type* → *uncertainty*) of {'calling new endpoint': 3, 'calling existing endpoint': 1, 'removing call': 1, 'updated caller version': 2, 'updated callee version': 2, 'updated version': 2, 'common call': 0} yielded the most promising results. We determined the nDCG for the top 5 positions to allow comparison across scenarios of different sizes.

Tracing Data. We implemented the two evaluation scenarios as microservice-based applications running on top of a Kubernetes cluster in the IBM Cloud. The Istio service mesh was in place to handle experiment traffic routing between the application's variants along with a Zipkin installation keeping track of service interactions. For every (sub-) scenario 1000 requests were generated.

5.2 Scenario 1: Revisiting the Sample Application

As a first scenario we use the example application shown in Fig. 1. Contrary to the next scenario, we do not cover a specific evaluation aspect here. However, this scenario involves all of the change types we identified, hence making it a useful baseline to assess the proposed heuristics.

Scenario. This scenario involves two sub-scenarios: *basic* and *delayed*. *Basic* executes the baseline variant of the application without modification, the canary variant involves added functionality and updated service versions. The *delayed* sub-scenario introduces a delay of $100ms$ at the *payment* service for the *canary* variant. This reflects an abnormally behaving *orders* service in the *canary* that multiplies the traffic towards the *payment* service causing it to overload, resulting in higher response times.

Relevance. For the basic scenario, the added calls to *product* and the updated versions of *frontend* and *orders* were classified as highly relevant (i.e., a relevance score of 4). For the delayed scenario, in addition, the call between *payment* and *orders* is classified as highly relevant. Relevance ratings for all scenarios are listed in our online appendix.

Table 1. $nDCG_5$ scores for all variations of the three heuristics across all evaluation scenarios. Scenario 1 with sub-scenarios basic and delayed (in the canary variant). Scenario 2 with four sub-scenarios: basic, a delay involving service j (canary), a delay involving service s (canary), and a combination of both delays (canary).

	Scenario 1		**Scenario 2**			
Heuristic	Basic	Delay	Basic	Delay j	Delay s	Combined
ST	0.89	0.93	0.91	0.83	0.87	0.76
ST Ext.	0.96	0.93	0.99	0.85	0.91	0.77
RTA	0.76	0.87	0.64	0.91	0.82	0.90
RTA Ext.	0.93	0.95	0.73	0.91	0.83	0.91
HYB	0.98	0.96	1.00	0.85	0.92	0.81
HYB Ext.	0.96	0.98	0.96	0.93	0.92	0.87

Results. Table 1 (Scenario 1) shows the nDCG scores of the three heuristics in their 6 variations for the *basic* and the *delayed* sub-scenarios. Scores are color-coded, the higher the score, the more intense the background color. The *hybrid* variations outperform the other heuristics, though some other approaches achieve high scores as well. *RTA* produces good results for the *delayed* sub-scenario. However, it only captures the "relevance" of the delayed fragments and ignores the high relevance of the added functionality. This is simply because there are no performance issues associated with these changes. The addition of *uncertainty* for the *RTA Ext* variant helps to compensate this flaw and leads to stronger scores for both sub-scenarios. Moreover, penalizing as a scoring factor turns out to have positive effects on the delayed sub-scenario. However, the standard *HYB* variant without penalties performs slightly better, though only by a whisker, e.g., by 0.005 on the combined score of both sub-scenarios for *HYB* and *HYB Ext.*

5.3 Scenario 2: Breaking Changes

The goal of the second scenario is to identify how the heuristics behave when dealing with more complex, cascading changes resulting in multiple version updates. This represents deployment scenarios and experiments dealing with multiple breaking API changes. Figure 1 (right) depicts its topological difference in which b is the experiment's target service.

Scenario. We split into multiple sub-scenarios involving simulated performance issues in the *canary* variant. In addition to the *basic* scenario, which contains multiple version updates and new services, we added two specific performance deviations: a delay at service h when calling service j (100 ms), and a delay at service s (200 ms) simulating a more complex request processing compared to the removed service pairs p, q, and r. As a fourth sub-scenario, we combined these two delays, making them active at the same time.

Relevance. For the *basic* sub-scenario, the version updates between b and c, b and f, f and m, and the added functionality for m calling s are rated as highly relevant. The delayed variants emphasize the changes introducing performance deviations.

Results. Similar to the running example, on average across all sub-scenarios, the *hybrid* heuristics perform best (see Table 1, Scenario 2). Some individual results on sub-scenarios provide valuable insights into the single heuristics' strengths and weaknesses. Keeping the *basic* results aside, *RTA* (in both variations) achieves an average nDCG score of 0.88, only topped by *HYB Ext*, which naturally inherited *RTA* functionality, with a score of 0.91. For the *basic* sub-scenario, the standard *HYB* performs best, almost reporting the perfect ranking with a score of 0.996, immediately followed by *ST Ext* with *uncertainty* involved (as propagation and scoring factor). Remarkably, the standard version of *ST* achieves a score of 0.91, also due to the fact that changes rated with high relevance are particularly "up high in the tree" (e.g., between b and f, and b and c) in this scenario. This enables this simple heuristic to come close to the best rankings.

5.4 Discussion

Combining the nDCG scores across all evaluation scenarios yields the highest (average) score of 0.94 for *HYB Ext*, a heuristic involving both uncertainty and a penalty mechanism in the scoring function. Interestingly, when diving deeper and distinguishing between (1) all basic scenarios and (2) all scenarios involving introduced performance issues we observe *HYB Ext* being not ranked first for both (1) and (2). Despite being superior for performance cases (2) with an average score of 0.93 and a gap of 0.03 to the second-best heuristic (i.e., *RTA Ext*), it is ranked third for non-performance cases, lacking a score of 0.03 to its leading standard *HYB* counterpart without penalty mechanism. As the performance cases dominate – 4 versus 2 non-performance cases – *HYB Ext* clearly benefits from the evaluation setup. This result is an indication that it would make sense to let developers or release engineers using our proposed tooling toggle between multiple (selected) heuristics which provide insights onto the application's state from different angles.

6 Limitations

One limitation of our approach is that the ranking quality evaluation was conducted on traces for self-generated scenarios. We mitigated this threat by covering two complex scenarios and combined them with sub-scenarios including simulated performance issues. A more thorough evaluation based on multiple real cases is desirable, and part of our future research. A further threat involves the relevance classification conducted by the authors of this paper. We classified all changes for all sub-scenarios on a scale from not relevant (0) to highly relevant (4). As the relevance is used as baseline for nDCG, these ratings have a direct effect on the resulting scores. Our online replication package allows inspecting how results change when relevance ratings are adjusted. Another threat involves the parameter calibration for the heuristics, which has a strong influence on the results. We mitigated this threat by performing thorough calibration runs with different parameter settings across all covered scenarios.

One limitation regarding the heuristics is that *RTA* variations only account for changes that impact the response time negatively. We focus on the total response time, ignoring that individual changes can have both positive and negative effects. However, our heuristics can be extended to cover this case as well.

Our evaluation focused solely on the ranking quality and left aside questions on how our approach would perform on industry-scale applications. We conducted a performance evaluation on the heuristic's execution behavior on self-generated difference graphs of multiple sizes and with various characteristics. First results are promising and show that the heuristics are able to cope with graphs consisting of thousands of nodes within seconds. However, detailed analysis are, also due to space reasons, out of scope for this paper and an evaluation on real instead of self-generated graphs is subject of future work.

7 Conclusion

We proposed an approach that analyzes request traces captured from distributed tracing systems to identify changes of microservice-based applications in the context of continuous experiments. Using heuristics, we rank these identified changes according to their potential impact on the experiment and the application's health state, with the goal of supporting decisions on whether to continue or abort the experiment. While previous work on experiment health assessment considers the services under test in isolation, which could skew the assessment as certain effects are left out, we focus on the topological level. We characterized a set of recurring topological change types consisting of fundamental patterns and more complex composed variants. We proposed three heuristics that operate on top of these characterized changes taking the concept of *uncertainty* into account. Our evaluation conducted on two case study scenarios demonstrated that the rankings produced by the heuristics are promising and could be a valuable resource for experiment health assessments. An comprehensive evaluation on how our approach performs on industry-scale applications is subject of future work.

References

1. Ates, E., et al.: An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019 (2019)
2. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Boston (2015). ISBN 0134049845
3. Davidovic, S., Beyer, B.: Canary analysis service. ACM Queue **16**(1) (2018). https://dl.acm.org/doi/10.1145/3190566
4. Froemmgen, A., Stohr, D., Koldehofe, B., Rizk, A.: Don't repeat yourself: seamless execution and analysis of extensive network experiments. In: Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2018) (2018)
5. Huhns, M.N., Singh, M.P.: Service-oriented computing: key concepts and principles. IEEE Internet Comput. **9**(1), 75–81 (2005)
6. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional, Bostan (2010). ISBN 0321601912
7. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inf. Syst. **20**(4), 422–446 (2002)
8. Kevic, K., Murphy, B., Williams, L., Beckmann, J.: Characterizing experimentation in continuous deployment: a case study on bing. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, pp. 123–132 (2017)
9. McSherry, F., Najork, M.: Computing information retrieval performance measures efficiently in the presence of tied scores. In: Macdonald, C., Ounis, I., Plachouras, V., Ruthven, I., White, R.W. (eds.) ECIR 2008. LNCS, vol. 4956, pp. 414–421. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78646-7_38
10. Newman, S.: Building Microservices, 1st edn. O'Reilly Media Inc., Newton (2015)

11. Sambasivan, R.R., et al.: Diagnosing performance changes by comparing request flows. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI 2011 (2011)
12. Santana, M., Sampaio, A., Andrade, M., Rosa, N.S.: Transparent tracing of microservice-based applications. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019 (2019)
13. Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., Stumm, M.: Continuous deployment at Facebook and OANDA. In: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE 2016, pp. 21–30, New York, NY, USA. ACM (2016)
14. Schermann, G., Schöni, D., Leitner, P., Gall, H.C.: Bifrost: supporting continuous deployment with automated enactment of multi-phase live testing strategies. In: Proceedings of the 17th International Middleware Conference, Middleware 2016, pp. 12:1–12:14, New York, NY, USA. ACM (2016)
15. Schermann, G., Cito, J., Leitner, P.: Continuous experimentation: challenges, implementation techniques, and current research. IEEE Softw. **35**(2), 26–31 (2018)
16. Schermann, G., Cito, J., Leitner, P., Zdun, U., Gall, H.C.: We're doing it live: a multi-method empirical study on continuous experimentation. Inf. Softw. Technol. **99**, 41–57 (2018)
17. Tang, C., et al.: Holistic configuration management at Facebook. In: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP), pp. 328–343, New York, NY, USA. ACM (2015)
18. Tarvo, A., Sweeney, P.F., Mitchell, N., Rajan, V., Arnold, M., Baldini, I.: CanaryAdvisor: a statistical-based tool for canary testing (Demo). In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), pp. 418–422, New York, NY, USA. ACM (2015)
19. Veeraraghavan, K., et al.: Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI 2016 (2016)
20. Xiao, Z., Wijegunaratne, I., Qiang, X.: Reflections on SOA and microservices. In: 4th International Conference on Enterprise Systems (ES) (2016)