Failure Root Cause Analysis for Microservices, Explained

Jacopo Soldani, Stefano Forti, Antonio Brogi

Failure Root Cause Analysis for Microservices, Explained

Jacopo Soldani(✉), Stefano Forti, and Antonio Brogi

University of Pisa, Pisa, Italy
{jacopo.soldani,stefano.forti,antonio.brogi}@unipi.it

Abstract. Determining the root causes of observed failures is a main issue in microservice-based applications. Unfortunately, available root cause analysis techniques do not focus on explaining how root failures actually caused the observed failure. On the other hand, the availability of such explanations would greatly help to pick adequate countermeasures, e.g., by introducing circuit breakers or bulkheads. We hence present a declarative root cause analysis technique, which can determine the cascading failures that possibly caused an observed failure, identifying also (or starting from) a root cause. We also introduce a prototype implementation of our technique, and we use it to assess our technique by means of controlled experiments.

1 Introduction

Microservices gained momentum in the software industry. For instance, Netflix and Spotify are already delivered as microservice-based applications [33]. This is because microservice-based applications are *cloud-native*, meaning that they are composed by loosely coupled services, which can be independently deployed and scaled to fully exploit the potentials of cloud computing [9].

Microservice-based applications are often composed by hundreds of services, which are typically replicated by instantiating multiple instances of each service. The multiple instances of the various different services in an application interact to deliver the end users' requests, possibly resulting in thousands of interactions happening at the same time. Service instances can fail, e.g., by returning error responses to their invokers, or not even answering since they suddenly crashed.

Whilst failing service instances can be promptly detected at runtime, understanding the possible root causes for a failing service instance is an offline task, which is inherently complex [28]. Did the service instance fail on its own? Did it instead fail in cascade, since it interacted with another failing service instance? Did the latter fail on its own or in cascade to some other service instance? Answering such questions is not easy, when you have possibly thousands of interactions among different service instances happening in parallel [33]. At the

Published by Springer Nature Switzerland AG 2022
D. Eyers and S. Voulgaris (Eds.): DAIS 2022, LNCS 13272, pp. 74–91, 2022.
https://doi.org/10.1007/978-3-031-16092-9_6

same time, answering the above questions is crucial to enact countermeasures and avoid the same failure cascade to happen again [28].

Existing root cause analysis techniques can help determining the service instances that may have failed first [28]. This is typically done by correlating the performance of the different service instances or the events they log, so as to determine the set of possible root causes. Identified root causes are also sometimes ranked by returning first those having higher chances to have caused the observed failure. However, there is no explanation of whether/how root causing failures propagated to other service instances, up to causing the failure observed on a service instance. Explanations —given as the possible failure cascades originating from an identified root causing failure— would enable intervening not only on the service that first failed, but also on those that failed in cascade [28]. They would enable, e.g., to equip intermediate services with circuit breakers enhancing the failure resilience of their instances [24], or to introduce bulkheads limiting failure propagations to only certain parts of an application [20].

We propose here a novel explainable root cause analysis technique, together with its prototype implementation and experimentation. Our technique automatically determines both the possible root causes for a failure observed on a service instance, and the failure cascades due to which the root causing failure possibly propagated up to that observed. It can also be used by restricting the possible root causes to a given set, hence enabling to explain the possible root causes identified with other existing techniques. In both cases, the explainable root cause analysis starts from the distributed logs of an application's service instances. Such logs are processed by means of declarative rules, which enable eliciting the interactions occurring among service instances, and determining whether a service instance failed on its own or in cascade, e.g., because it interacted with another failing service instance.

We also present `yRCA`, an open source prototype implementation of our explainable root cause analysis technique. We show how we used `yRCA` to run controlled experiments assessing our technique, based on an existing chaos testbed. The results of our experiment show that our technique can effectively determine the possible root causes and their explanations in 99.74% of the cases, whilst also keeping the number of returned explanations low enough to be counted on one hand. Our experiments also show that `yRCA` already achieves good time performances, with a processing time that is low on average, especially if we consider that root cause analysis is a batch process to be enacted offline [28].

The paper is organised as follows. Section 2 motivates our work. Sections 3 and 4 introduce our explainable root cause analysis technique and its prototype implementation, respectively. Section 5 presents some controlled experiments assessing our technique. Finally, Sects. 6 and 7 discuss related work and draw some concluding remarks, respectively.

2 Motivating Scenario

Consider *Sock Shop* [34], an open source application simulating an e-commerce website selling socks. The microservice-based architecture of *Sock Shop* is dis-

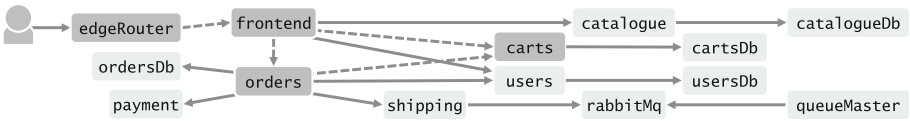


Fig. 1. *Sock Shop*'s microservice-based architecture. Darker nodes and dashed arcs highlight the portion of *Sock Shop* discussed in our motivating example.

played in Fig. 1. Clients connect to *Sock Shop* through an `edgeRouter`, which redirects clients' requests to the possibly multiple instances of the application's `frontend`. The latter displays a graphical user interface for e-shopping socks, backed by the microservices managing the `catalogue` of socks, the application `users`, and the users' `carts` and `orders`. Each of such microservices interacts with its own database to persist data, viz., `catalogueDb`, `usersDb`, `cartsDb`, and `ordersDb`. `orders` also interact with `carts` and `payment` to allow placing orders by simulating the online payment of the socks in a cart, and with `shipping` to simulate the actual shipping of an order. This is done with the microservice `shipping` placing to-be-shipped orders in a message queue (viz., `rabbitMq`), which is consumed by `queueMaster` to simulate their actual shipping.

Consider a running deployment of *Sock Shop*, with two replicated instances of each microservice, and focus on the highlighted portion of *Sock Shop* in Fig. 1. Suppose that an instance of `carts` fails (e.g., because of an internal error) and starts returning error responses to its clients. Suppose also that `frontend`'s instances can tolerate the failure of `carts`' instances, e.g., by caching carts. Suppose instead that `orders`' instances fail when `carts` replies with error responses, becoming unable to process the requests from `frontend`'s instances. When this happens, we have cascading failures in `frontend` as well, due to which *Sock Shop*'s end users cannot place orders.

For *Sock Shop* to get back fully working, application operators must identify the internal failure of an instance of `carts` as the root cause of the failures in `frontend`'s instances, as well as that such root causing failure propagated to `frontend` through `orders`. This would enable first recovering the failing instance of `carts`, e.g., by restarting it, which would then result in the instances of `orders` and `frontend` getting back fully working as well. Also, by identifying the failure cascades that made `frontend`'s instances unable to place orders, application operators could operate *only* on such cascades to avoid this to happen again. For instance, they could introduce a circuit breaker enabling `orders` to tolerate the failure of `carts`' instances, whilst not intervening on `frontend`, which can already tolerate the failure of `carts`.

There exist techniques for identifying the possible root causes of functional failures in microservice-based applications, e.g., [8,12,13,18,21,22,26,31]. However, they only identify the root causing microservice without stating how its failure propagated to other microservices. Understanding failure propagation still requires to manually inspect the distributed logs of an application, or its distributed traces, if the application is instrumented to feature distributed tracing.

Due to the number of microservices in an application, their complex interactions, and considering that microservices can be replicated over multiple instances, the resulting process is cumbersome, error-prone, and time consuming [28].

To this end, we provide a novel technique for identifying the failure cascades that can have possibly caused a failure in a microservice instance. Our technique inputs the distributed application logs, and it can automatically identify the possible root causing failures and how they propagated to a failing microservice instance. Our technique can also be used to complement the existing root cause analysis techniques, if any is in place, by identifying the possible cascades explaining how the identified root causing failure propagated and caused that observed on a failing microservice instance.

3 Declarative Failure Root Cause Analysis

In this section, we describe the declarative Prolog[1] methodology that enables determining explanations for failure root causes, through interaction-based analyses, which rely on simple logging information.

Logs and Interactions. First, application logs are modelled as facts like

```
log(SName, SInstance, Timestamp, Event, Message, Severity).
```

where the name `SName` and the instance identifier `SInstance` of the logging service are followed by the log `Timestamp`, the type of the logged `Event`, the associated log `Message` (if any), and its `Severity` level. Our methodology currently handles the following types of event:

- `internal`, which denotes logs related to the internal business logic of the considered service,
- `sendTo(DstService, SessionId)`, which denotes that a request was sent to an instance of `DstService` with an associated `SessionId`,[2]
- `received(SessionId)`, which denotes reception of a message by an instance of the destination service within the interaction identified by `SessionId`,
- `timeout(DstService, SessionId)`, which denotes that the interaction started towards `DstService`, identified by `SessionId`, incurred in a timeout, and
- `errorFrom(DstService, SessionId)`, which denotes that the destination service replied with an error code within the interaction identified by `SessionId`.

Finally, logged severity levels are expressed according to the Syslog standard [7]:

[1] A Prolog program is a finite set of *clauses* of the form `a :- b1, ..., bn.` stating that `a` holds when `b1` $\wedge \cdots \wedge$ `bn` holds, where `n` ≥ 0 and `a`, `b1`, ..., `bn` are atomic literals. The logical or of two literals `b1` $\vee$ `b2` can be expressed as `b1; b2`. Clauses with empty condition are also called *facts*. Prolog variables begin with upper-case letters, lists are denoted by square brackets, and negation by `\+`.

[2] To easily identify messages pertaining to the same interaction, we assume that the code performing a request generates fresh session identifiers for each interaction.

```
severity(emerg, 0).   severity(err, 3).        severity(info, 6).
severity(alert, 1).   severity(warning, 4).    severity(debug, 7).
severity(crit, 2).    severity(notice, 5).
```

Note that Syslog severity levels can be mapped onto other existing industry standards (e.g., Log4J), and compared one another via a predicate like

```
moreSevere(Sev1,Sev2) :- severity(Sev1,A), severity(Sev2,B), A<B.
```

which holds when the severity level identified by `Sev1` is more severe than the one identified by `Sev2`.

Based on this simple modelling, we can identify any completed interaction between instance `I` of service `SI` and instance `J` of service `SJ`, that happened between time `Ts` and time `Te`. Predicate `interaction/5` infers that instance `I` of service `SI` performed a request towards instance `J` of service `SJ` and that, in turn, instance `J` of `SJ` logged reception of such request. Such interaction is identified by its unique session `Id` and happened at time `Tr`, between `Ts` and `Te`:

```
interaction(Id,(SI,I),(SJ,J),Ts,Te) :-
  log(SI,I,Ts,sendTo(SJ,Id),_,_), log(SJ,J,Tr,received(Id),_,_),
  Ts < Tr, Tr < Te.
```

Dually, predicate `nonReceivedRequest/5` covers the case in which a request sent by instance `I` of `SI` incurred in a timeout event and its reception was not logged by any instance of `SJ` between `Ts` and `Te`:

```
nonReceivedRequest(Id,I,SJ,Ts,Te) :-
  log(SI,I,Ts,sendTo(SJ,Id),_,_), log(SI,I,Te,timeout(SJ,Id),_,_),
  \+ ( log(SJ,_,Tr,received(Id),_,_), Ts =< Tr, Tr =< Te ).
```

By relying on `interaction/5`, predicate `failedInteraction/5` identifies interactions that – despite being correctly received at the destination service – failed either due to a logged error or to an expired timeout:

```
failedInteraction(Id,(SI,I),(SJ,J),Ts,Te) :-
  errorInteraction(Id,(SI,I),(SJ,J),Ts,Te) ;
  timedOutInteraction(Id,(SI,I),(SJ,J),Ts,Te).
```

On the one hand, `errorInteraction/5` identifies that the invoked service instance `J` of `SJ` terminated the interaction identified by `Id` by returning an error response to the source service instance `I`:

```
errorInteraction(Id,(SI,I),(SJ,J),Ts,Te) :-
  log(SI,I,Te,errorFrom(SJ,Id),_,_),
  interaction(Id,(SI,I),(SJ,J),Ts,Te).
```

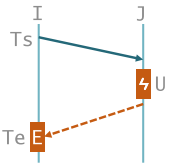
On the other hand, predicate `timedOutInteraction/5` handles the case in which the interaction was interrupted by the invoking service instance `I` of `SI`, since the corresponding request's timeout expired:

```
timedOutInteraction(Id,(SI,I),(SJ,J),Ts,Te) :-
  log(SI,I,Te,timeout(SJ,Id),_,_),
  interaction(Id,(SI,I),(SJ,J),Ts,Te).
```

Explanations. Predicate `causedBy/3` is the core that our methodology exploits to determine explanations for root causes. A call to `causedBy(Log, Explanation, RootCause)` inputs an event `Log` to be explained and recursively builds an `Explanation`, represented as a list of logs, until it determines `RootCause` as the name of the service that started the failure cascade. Predicate `causedBy/3` distinguishes 8 cases – 5 recursive and 3 base cases – corresponding to different possible cascading or root failures, respectively. For each case, we illustrate the Prolog code and offer a graphical sketch to epitomise recursive cases.

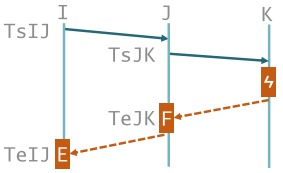
The first case (lines 1–7) infers that event `E` logged by instance `I` of service `SI` (line 2), currently being explained, has been caused by an *internal error of the invoked service instance* `J` of `SJ`. Event `E` is either an error or a timeout (line 3), resulting from an interaction between `I` and `J` that failed or timed-out between time `Ts` and `Te` (line 4). If `J` logged an internal error more severe than a `warning` in the same time period (line 5–6), then the `Log` currently being explained is added to the explanation (line 1), and `causedBy/3` recurs to possibly explain the internal error of `J` (line 7).

```
1  causedBy(Log,[Log|Xs],Root) :-
2    Log=log(SI,I,T,E,M,Sev),
3    (E=errorFrom(SJ,Id);E=timeout(SJ,Id)),
4    failedInteraction(Id,(SI,I),(SJ,J),Ts,Te),
5    L=log(SJ,J,U,internal,MJ,SevJ),
6    moreSevere(SevJ,warning), Ts =< U, U =< Te,
7    causedBy(L,Xs,Root).
```



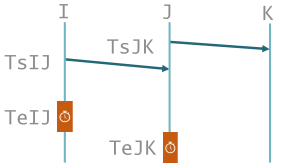
The second case (lines 8–15) infers that event `E` logged by instance `I` of service `SI` (line 9), currently being explained, has been caused by a cascading *failed interaction of the invoked service instance* `J` of `SJ`. Event `E` is either an error or a timeout (line 10) resulting from an interaction between `I` and `J` that failed between time `TsIJ` and `TeIJ` (line 11). In turn, such failure has been caused by a failed interaction of `J` with a third service between time `TsJK` and `TeJK` (line 12), within `TsIJ` and `TeIJ` (line 13). If `J` logged an event `F` at time `TeJK` (line 14) more severe than a `warning` (line 15), then the `Log` currently being explained is added to the explanation (line 8), and `causedBy/3` recurs to possibly explain the log `L` related to event `F` (line 16).

```
8  causedBy(Log,[Log|Xs],Root) :-
9    Log=log(SI,I,T,E,M,Sev),
10   (E=errorFrom(SJ,Id);E=timeout(SJ,Id)),
11   failedInteraction(Id,(SI,I),(SJ,J),TsIJ,TeIJ),
12   failedInteraction(_,(SJ,J),(_,_),TsJK,TeJK),
13   TsIJ =< TsJK, TeJK =< TeIJ,
14   L=log(SJ,J,TeJK,F,MJ,SevJ),
15   moreSevere(SevJ,warning),
16   causedBy(L,Xs,Root).
```



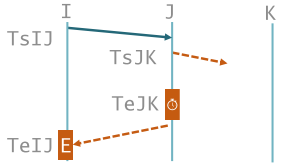
The third case (lines 17–23) infers that event `E` logged by instance `I` of service `SI` (line 18), currently being explained, has been caused by the *timed-out interaction of the invoked service instance* `J` of `SJ`. Event `E` is a timeout resulting from an interaction between `I` and `J` timed out at time `TeIJ` (line 19). In turn, such timeout has been caused by a timed out interaction of `J` with a third service `SK` between time `TsJK` and `TeJK` (line 20), started before the previous interaction and timed out afterwards (line 21). If `J` logged a timeout event at time `TeJK` related to an interaction `IdJK` with `SK` (line 22), then the `Log` currently being explained is added to the explanation (line 17), and `causedBy/3` recurs to possibly explain the timeout of interaction `IdJK` (line 23).

```
17 causedBy(Log,[Log|Xs],Root) :-
18   Log=log(SI,I,T,timeout(SJ,Id),M,Sev),
19   timedOutInteraction(Id,(SI,I),(SJ,J),_,TeIJ),
20   timedOutInteraction(_,(SJ,J),(SK,_),TsJK,TeJK),
21   TsJK =< TeIJ, TeIJ < TeJK,
22   L=log(SJ,J,TeJK,timeout(SK,IdJK),MJ,SevJ),
23   causedBy(L,Xs,Root).
```



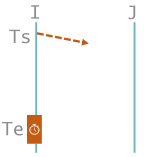
The fourth case (lines 24–31) infers that event `E` logged by instance `I` of service `SI` (line 25), currently being explained, has been caused by the *unreachability of a service called by the invoked service instance* `J` of `SJ`. Event `E` is either an error or a timeout (line 26) resulting from a failed interaction `Id` between `I` and `J` between time `TsIJ` and `TeIJ` (line 27). In turn, such an event has been caused by a request of `J` towards service `SK`, happening between `TsIJ` and `TeIJ`, which was never received by any instance of `SK` (lines 28–29). If `J` logged a timeout event at time `TeJK` related to interaction `IdJK` with `SK` (line 30), then the `Log` currently being explained is added to the explanation (line 24), and `causedBy/3` recurs to possibly explain the time out of interaction `IdJK` (line 31).

```
24 causedBy(Log,[Log|Xs],Root) :-
25   Log=log(SI,I,T,E,M,Sev),
26   (E=errorFrom(SJ,Id);E=timeout(SJ,Id)),
27   failedInteraction(Id,(SI,I),(SJ,J),TsIJ,TeIJ),
28   nonReceivedRequest(IdJK,J,SK,TsJK,TeJK),
29   TsIJ =< TsJK, TsJK =< TeIJ,
30   L=log(SJ,J,TeJK,timeout(SK,IdJK),MJ,SevJ),
31   causedBy(L,Xs,Root).
```



The fifth case (lines 32–35) infers that event `E` logged by instance `I` of service `SI` (line 33), currently being explained, has been caused by the *unreachability of the invoked service instance* `J` of `SJ`. Indeed, `I` incurred in a timeout during an interaction `Id` with `SJ` (line 33), which was caused by a request that was never received at any instance of `SJ` (line 34). Then, the `Log` currently being explained is added to the explanation (line 32), and `causedBy/3` recurs to possibly explain an abducted piece of knowledge, i.e. that `SJ` was unreachable (line 35).

```
32 causedBy(Log,[Log|Xs],Root) :-
33   Log = log(SI,I,T,timeout(SJ,Id),M,Sev),
34   nonReceivedRequest(Id,I,SJ,_,T),
35   causedBy(unreachable(SJ),Xs,Root).
```



The sixth case (line 36) explains an `internal` event `R` logged by a service, identifying the service itself as the `Root` cause for `R`. Recursion ends.

```
36 causedBy(Log,[Log],Root) :- Log = log(Root,R,T,internal,M,Sev).
```

The seventh case (line 37) explains an abducted `unreachable(Root)` fact, identifying that such a service was temporarily unreachable as it previously logged some information. Recursion ends.

```
37 causedBy(Log,[Log],Root) :- Log = unreachable(Root), log(Root,_,_,_,_,_).
```

The last case (lines 38–39) explains an abducted `unreachable(Root)` fact (line 34), identifying that such a service never logged any information (line 39). Recursion ends, by adding the fact that `Root` was possibly never started (line 40).

```
38 causedBy(unreachable(Root),[Log],Root) :-
39   \+ log(Root,_,_,_,_,_),
40   Log = neverStarted(Root).
```

Last, but not least, `xfail(Event,Explanations,RootCause)` exploits `causedBy/3` to determine all distinct `Explanations` starting from the service `RootCause`, as in:

```
xfail(Event,Explanations,RootCause) :-
  findall(E,distinct(causedBy(Event,E,RootCause)),Explanations).
```

It is worth noting that, thanks to Prolog resolution mechanisms, our methodology permits instantiating the `RootCause` parameter to a specific service name so to restrict the obtained explanations only to those that have such a service as the failure cascade root cause. If, conversely, `RootCause` is left unbound, it determines all explanations for all possible values that can be unified with `RootCause`. This enables using our methodology both as an explainer working in pipeline with other existing tools for root caused identification and as a standalone tool.

4 Prototype Implementation

We developed an open source prototype of our explainable analysis technique, called yRCA.[3] yRCA embeds the Prolog reasoner presented in the previous section in a Python-based command-line tool, which can be run as follows:

```
python3 yrca.py [-r S] [-v] EVENT LOGS TEMPLATES
```

where EVENT and LOGS are two JSON files containing the failure event to be explained and a dump of the distributed logs of all service instances in an application. In both JSON files, log entries are expected to be in GELF (*Graylog Extended Log Format* [5]). TEMPLATES is instead a YAML file specifying the templates to parse log messages, with each template being a regular expression to match log messages to determine whether they correspond to client-side events (viz., request sent, successful/error response received, or timeout expired) or server-side events (viz., request received, response sent). Finally, option -r S enables focusing on explanations having S as the root causing service, e.g., to explain how its failure – identified with another root cause analysis technique – caused that in EVENT. Option -v instead allows running yRCA in *verbose* mode, namely by printing all possible explanations, rather than grouping them based on their structure, as yRCA does by default.

An example of output returned by yRCA is shown hereafter, with possible explanations for the failure mentioned in our motivating scenario (Sect. 2):

```
[0.615]: edgeRouter: Error response (code: 500) received from frontend
(request_id: [<requestId>])
   -> frontend: Error response (code: 500) received from orders (request_id:
      [<requestId>])
   -> orders: Failing to contact carts (request_id: [<requestId>]). Root
      cause: <exception>
   -> carts: unreachable
[0.385]: edgeRouter: Error response (code: 500) received from frontend
(request_id: [<requestId>])
   -> frontend: Failing to contact carts (request_id: [<requestId>]). Root
      cause: <exception>
   -> carts: unreachable
```

Note that, by default, yRCA groups the possible explanations based on the structure of the failure cascade, and it ranks the different explanations based on the frequency with which they occur in all identified failure cascades —with such frequency indicated between square brackets at the beginning of each explanation. The idea is that the more frequent is an explanation, the higher is the probability that it corresponds to the true explanation for an observed failure. This is inspired by other existing analysis techniques, which rank the identified root causes by giving higher ranks to those found with a higher rate [28].

[3] https://github.com/di-unipi-socc/yrca.

5 Evaluation

To assess the practical applicability of our root cause analysis technique, we run `yRCA` in controlled experiments.[4] Their objective was to evaluate the performances of our technique in determining the failure cascades that may have caused an observed failure, namely whether the true cause is among those returned, how many possible explanations were returned, and the elapsed time.

In our experiments, we exploited the Chaos Echo testbed [27] to obtain a reference application. Chaos Echo enables deploying interconnected services to mirror the architecture of an existing application, while replacing each of its services with a Chaos Echo service. A Chaos Echo service simulates the behaviour of an existing service by interacting with its backend services (if any) to process incoming requests, and by possibly failing in doing so. Whenever it receives a request, it interacts with a randomly selected subset of its backend services, each invoked with a given probability. The Chaos Echo service forwards them the incoming request's message and waits for their answer. If any of the invoked backend services returns an error response, or if a request timeout expires, the Chaos Echo service considers the interaction as failed. It then fails in cascade, by replying to the request under processing with an error response. A Chaos Echo service may also fail on its own, with a given probability, either returning an error response (even if all its backend services successfully replied to its requests) or by suddenly crashing, hence not replying at all. By differently configuring the Chaos Echo services in a reference application, we can control how their services interact, fail, and propagate failures. This, together with the workload generator available in the Chaos Echo testbed, enable assessing tools for analysing failures in multi-service applications, like `yRCA`.

The reference application we used in our experiments mirrors *Sock Shop* (Fig. 1), by replacing each of its components by a Chaos Echo service. The Chaos Echo service replacing `edgeRouter` was then configured to always invoke `frontend` and to never fail on its own, but only in cascade to the service it interacts with, viz., `frontend`. Our objective was indeed to assess `yRCA`'s ability to determine the root causing failures and the cascades that resulted in failures observed on `edgeRouter`. All other Chaos Echo services were then differently configured to analyze the performances of `yRCA` when varying four different parameters, viz., (a) end-user load, (b) service interaction rate, (c) failure cascade length, and (d) service failure rate. More precisely:

(a) We configured all services (but `edgeRouter`) to invoke their backend services with probability 0.5. We also set them to never fail on their own, with the only exception of `carts`, which was set to fail with probability 0.5. We then varied the end-user load from 1 to 100 req/s.
(b) We varied the probability with which all services (but `edgeRouter`) invoked their backend services from 0.1 to 1. We also set them to never fail on their

[4] The sources of the controlled experiments are publicly available at https://github.com/di-unipi-socc/yrca/tree/main/data/experiments/sock-echo.

own (except for carts, which was set to fail with probability 0.5) and we fixed the end-user load to 10 req/s.

(c) We configured all services (but edgeRouter) to invoke their backend services with probability 0.5 and set the end-user load to 10 req/s. We then varied the length of the considered failure cascade by considering the different cases of frontend, orders, shipping, or rabbitMq being the *only* services set to fail on their own with probability 0.5. This enabled us to generate failure cascades of length 1, 2, 3, or 4, respectively.

(d) We configured the services to invoke their backend services with probability 0.5 and set the end-user load to 10 req/s. We then set all services (but edgeRouter) to fail with a probability varying from 0.1 to 1.

In all cases, to also account for service instances, all services (but edgeRouter) were set to be replicated over two instances. Overall, we hence always had 27 deployed service instances.

We run the differently configured deployments to generate logs, ensuring that each deployment was generating at least 200 failures in edgeRouter. We then run yRCA to explain a random sample of 200 edgeRouter's failures in the logs of each case, repeating the run 5 times for each failure, so as to measure the average time yRCA took to explain a failure in each case. As a result, we observed that the results returned by yRCA contained the true root cause and the corresponding explanation in 99.74% of the times. The effectivenss of our technique however not only depends on this, but also on the number of returned false positives, viz., failure cascades considered to have possibly caused the observed failure, even if this was not the case. False positives should be kept low, as they require application operators to waste time and resources in unnecessarily checking them [28]. We therefore measured the average number of failure cascades identified by yRCA, one being the right solution and the other being false positives.[5]

Number of Identified Failure Cascades. The results of our measurements are shown in Fig. 2. We can readily observe that in cases (a–c), where there was only one service set to possibly fail (and to cause subsequent failure cascades), yRCA correctly determined only one possible root cause. The latter effectively corresponded to the known ground truth in all the three cases, viz., carts in cases (a) and (b), and each service set to fail in each different cascade for case (c). yRCA also identified a number of possible explanations originating from carts that slightly increased when we increased (a) the load rate, (b) the rate with which each service was invoking its backend services, and (c) the length of the failure cascade. This is mainly because the increasing load/interactions in (a–b) resulted in an increasing number of service interactions, whilst in (c) we had an increasing number of services failing in cascade. For this reasons, (a–c) resulted in an increasing number of possible paths for the root causing failure to propagate up to edgeRouter, which is reflected by the plots (a–c) in Fig. 2.

The results were instead quite different when we increased the probability with which each service (but edgeRouter) could have failed, as shown in Fig. 2(d).

[5] All experiments reported in this section were executed on a Ubuntu 20.04.3 LTS virtual machine, with four vCPUs and 32 GB of RAM.

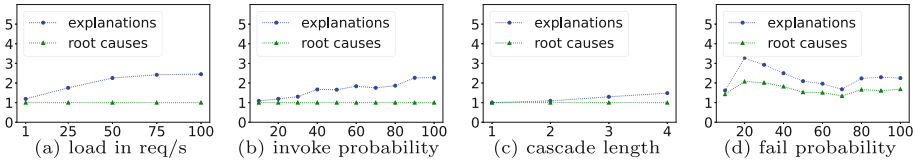


Fig. 2. Average number (y-axis) of identified explanations and root causes.

In this case, all instances of the services in Fig. 1 (but `queueMaster`) not only get invoked to process an end-user request received by `edgeRouter`, but they may also fail on their own with increasing probability. Their failures may then propagate and cause that observed on `edgeRouter`. Even in such a complex scenario, with all instances of 12 services possibly being the root cause of an observed failure, the average numbers of root causes and explanations returned by `yRCA` were (in the worst case) around two and three, respectively. Out of those, one always corresponded to the true root cause and failure cascade that happened in our reference application deployment.

The above discussed experiments (a–d) show that `yRCA` not only effectively identified the failure cascades that caused an observed failure, but also that it was able of restricting such cascades to quite a few. This hence reduces the number of false positives returned by `yRCA`, which is a plus when enacting root cause analysis in applications composed of many interacting microservices [28].

Average Processing Time. Figure 3 shows the average time required by `yRCA` to explain each failure in each experiment, normalised in milliseconds for processing a megabyte of logs. We can observe that the average processing time depends on how much services interacts. Indeed, elapsed time significantly grew with (a) the load rate, whose increase results more service interactions due to a higher number of end users' requests to be processed, and with (b) the probability of each service invoking its backend services. This is to be expected, given that our Prolog reasoner first identifies and classifies service interactions, to then process classified interactions to reconstruct possible failure cascades (Sect. 3). It is anyhow worth noting that, even in the cases of heavy load, `yRCA` took 104.04 s to process 381.21 MBs of logs. Such an amount of time is acceptable for an offline task as that of failure root cause analysis [37]. It is also much less than that we would need to elicit the failure cascades that may have caused a failure by manually inspecting the same logs, even in the case when the possible root causing failures have already been identified with some existing technique (like those discussed in Sect. 6).

Figure 3 also shows that the average processing time instead kept stable when we increased (c) the length of failure cascades or (d) the services' failure rate. This suggests that our root cause analysis is independent from how long is a failure cascade or how many services failed, as it would require the same processing time. This is a desiderata when having big enterprise applications where a failure cascade may involve tens of services [30].

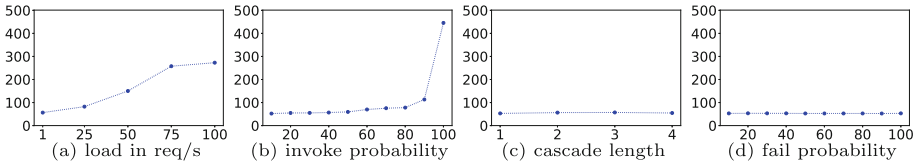


Fig. 3. Average elapsed time (y-axis) for each experiment, in ms/MB.

6 Related Work

Various techniques have been proposed to identify the possible root causes of failures in multi-service applications [28]. This is typically done by relying on applications being instrumented to feature distributed tracing or to monitor specific Key Performance Indicators (KPIs) on their services. For instance, Zhou *et al.* [37] and Guo *et al.* [6] propose two methodologies to systematically identify the root cause of a failure observed on the frontend of an application, based on manually inspecting its distributed traces with the support of visualisation tools. Similarly to our technique, their methodologies enable identifying not only the possible root causing failures, but also the cascades that made such failures propagate up to that observed on the application frontend. They however differ from our technique, since we enable identifying the possible root causes for failures happening on any service in an application, whilst also fully automating the root cause analysis.

CloudDiag [18], TraceAnomaly [13], MonitorRank [8], and MicroHECL [12] are other examples of distributed tracing-based root cause analysis techniques, yet fully automating the analysis. CloudDiag [18] and TraceAnomaly [13] directly analyse the distributed traces and consider as possible root causes for a failure those services whose response time was anomalous. MonitorRank [8] and MicroHECL [12] instead process the distributed traces to obtain a graph representing the services forming the application and their interactions, which they then visit guided by the correlation between service performances. The services corresponding to the nodes where their visits stop are considered as possible root causes for the considered failure. A first difference between the above techniques (but MicroHECL [12]) resides in the fact that they focus on analysing the possible root causes of failures happening on the application frontend, whilst we enable analysing those for failures happening on any service. In addition, all the above techniques differ from ours since they require application to feature distributed tracing, and since they return the possible root causes for a failure, but without explaining how they propagated and caused the observed failure. Our technique can hence complement their results, by not only returning the possible root causes of an observed failure, but also the cascades that made root causing failures propagate to that observed.

Similar considerations apply to the root cause analysis techniques requiring to instrument applications to monitor specific KPIs on their services. For instance, ϵ-diagnosis [26], PAL [21], Wang *et al.* [31], and FChain [22] enable determining

the possible root causes for a service's failure, based on the correlation between anomalous KPI values monitored on the failing service and on other services. They all differ from our technique since, even if identifying the possible root causes for an observed failure (which must be a frontend failure in the case ϵ-diagnosis [26], PAL [21], and FChain [22]), they are not providing explanations on how the root causing failures propagated to that observed. We instead enable identifying the failure cascades that caused a failure observed on any service. We can thus complement the results obtained with ϵ-diagnosis [26], PAL [21], Wang *et al.* [31], or FChain [22], by allowing to determine how the root causing failures – identified with such techniques – propagated and caused that observed.

MicroRCA [35], Wu *et al.* [36], Sieve [30], and Brandón *et al.* [2], and DLA [25] exploit monitored KPIs to drive the search for the possible root causes of a failure in a graph-based modelling of the architecture of an application. The latter is automatically reconstructed by MicroRCA [35], Wu *et al.* [36], Sieve [30], and Brandón *et al.* [2] from monitored KPIs themselves, and it is instead an input for DLA [25]. Despite relying different methods to visit the graph, they can all effectively determine the possible root causes for an observed failure. At the same time, they all differ from our technique since they return the possible root causing failures, without eliciting how such failures propagated and caused that observed. We instead enable identifying the whole failure cascades that caused an observed failure, also allowing to explain how the root causing failures identified with MicroRCA [35], Wu *et al.* [36], Sieve [30], Brandón *et al.* [2], or DLA [25] caused that observed.

Similar considerations apply to CauseInfer [4], Microscope [10], Qiu *et al.* [23], CloudRanger [32], MS-Rank [14], AutoMap [15], MicroCause [17], FacGraph [11], and LOUD [16]. They all exploit monitored KPIs to infer a causality graph, whose nodes model the services forming an application and whose arcs model causal relationships between the performances of its services. Most of them then exploit KPIs to identify the possible root causes of a failure by visiting the causality graph, with CauseInfer [4], Microscope [10], and Qiu *et al.* [23] enacting a KPI-driven BFS, whereas CloudRanger [32], MS-Rank [14], AutoMap [15], MicroCause [17] enact a random walk similar to that of MonitorRank [8]. FacGraph [11] and LOUD [16] instead analyze the graph structure to determine the possible root causes for a failure. Again, the main difference between such techniques and ours resides in explainability. The above techniques indeed effectively identify the possible root causes for an observed failure, without explaining how the root causing failures propagated to that observed. Our technique instead determines the failure cascades that may have possibly caused an observed failure, hence also enabling to complement the results that can be obtained with the above discussed techniques.

Finally, it is worth positioning our work with respect to Aggarwal *et al.* [1] and our previous work [29], which both process the logs produced by the services in an application, instead of requiring it to feature distributed tracing or to get instrumented with monitoring probes. Aggarwal *et al.* [1] model the logs of the services forming an application as multivariate time series, and it then

exploits Granger causality tests to derive a causality graph. In the latter, nodes model services, whilst arcs model causal dependencies among the errors logged by services. The causality graph is then visited by enacting a random walk similar to MonitorRank [8], in order to determine the highest probable root cause for a failure observed on the application frontend. Aggarwal *et al.* [1] hence differ from our technique since they return one possible root cause for a frontend failure, without explaining how such root causing failure propagated to the application frontend. We instead enable determining the possible root causes for failures observed on any service, while also eliciting the failure cascades that may have caused root causing failures to propagate to the observed ones.

In this perspective, the root cause analysis technique we proposed in our previous work [29] is closer to that in this paper, given that it identifies the possible root causes and explains how they propagated to cause an observed failure. Our previous work [29] however relies on a specification of the application architecture and of the failure behaviour of each of the service therein, given by associating each service to its fault-aware management protocol [3]. We indeed exploited such specification to drive the search for failure cascades in the application logs. The technique presented in this paper hence differs from that in our previous work [29], given that we now directly process the application logs, without requiring any specification of the application.

In summary, to the best of our knowledge, ours is the first explainable root cause analysis technique, which not only determines the possible root causes for a failure, as typically done in literature, but also the cascades due to which the root causing failures propagated and caused that observed. It is the first doing it in a fully automated manner, and without requiring other inputs than the logs produced by the services forming an application. Our technique can also complement the results obtained with other existing techniques, by explaining how the root causing failures they identify propagated and caused that observed.

7 Conclusions

We presented an explainable technique for determining the possible root causes of cascading failures in any microservice-based application, provided that its services suitably log their interaction and failure events (Sect. 5). Our technique can be used to determine the failure cascades that possibly caused an observed failure, either also eliciting the possible root causes or starting from a given set of possible root causes. It can hence complement existing root cause analysis techniques, providing explanations of how the root causing failures they identify propagated and caused that observed.

We also presented a prototype implementation of our technique, called `yRCA`, which we used to assess it based on controlled experiments run on an existing chaos testbed. The results of our experiments showed that `yRCA` features good time performances, and that it effectively determined the root cause of a failure in 99.74% of the cases. This happened whilst returning around 3 possible explanations in the worst case. `yRCA` hence kept the number of false negatives

low, thus limiting the efforts hat should be spent by application administrators in checking failure cascades that did not truly caused an observed failure.

The failure cascades explaining an observed failure can help application administrators in identifying where to enact suitable countermeasures (e.g., circuit breakers and bulkheads [20]) to avoid the occurrence of those failure cascades. One natural direction for future work is the prototyping of a tool supporting the visualization of failure cascades explaining observed failures, together with suggestions of possible countermeasures.

Another interesting future work direction is experimenting our technique on industrial applications, based on different chaos testing approaches (e.g., Netflix's Chaos Monkey [19]). We also plan to extend the scope of our explainable root cause analysis to deal with incomplete logs, e.g., in case the logging driver fails or a service instance gets suddenly killed without flushing all its logs.

References

1. Aggarwal, P., et al.: Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In: Hacid, H., et al. (eds.) ICSOC 2020. LNCS, vol. 12632, pp. 137–149. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76352-7_17
2. Brandón, A., et al.: Graph-based root cause analysis for service-oriented and microservice architectures. J. Syst. Soft. **159**, 110432 (2020). https://doi.org/10.1016/j.jss.2019.110432
3. Brogi, A., et al.: Fault-aware management protocols for multi-component applications. J. Syst. Softw. **139**, 189–210 (2018). https://doi.org/10.1016/j.jss.2018.02.005
4. Chen, P., et al.: Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: INFOCOM 2014, pp. 1887–1895. IEEE (2014). https://doi.org/10.1109/INFOCOM.2014.6848128
5. Graylog Extend Log Format: Graylog (2022). https://www.graylog.org/
6. Guo, X., et al.: Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: ESEC/FSE 2020, pp. 1387–1397. ACM (2020). https://doi.org/10.1145/3368089.3417066
7. IETF: The Syslog protocol. RFC 5424, Network Working Group (2009)
8. Kim, M., et al.: Root cause detection in a service-oriented architecture. SIGMETRICS Perform. Eval. Rev. **41**(1), 93–104 (2013). https://doi.org/10.1145/2494232.2465753
9. Kratzke, N., Quint, P.: Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. J. Syst. Soft. **126**, 1–16 (2017). https://doi.org/10.1016/j.jss.2017.01.001
10. Lin, J., Chen, P., Zheng, Z.: Microscope: pinpoint performance issues with causal graphs in micro-service environments. In: Pahl, C., Vukovic, M., Yin, J., Yu, Q. (eds.) ICSOC 2018. LNCS, vol. 11236, pp. 3–20. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03596-9_1
11. Lin, W., et al.: FacGraph: frequent anomaly correlation graph mining for root cause diagnose in micro-service architecture. In: IPCCC 2018, pp. 1–8. IEEE (2018). https://doi.org/10.1109/PCCC.2018.8711092

12. Liu, D., et al.: MicroHECL: high-efficient root cause localization in large-scale microservice systems. In: ICSE-SEIP 2021, pp. 338–347. IEEE (2021). https://doi.org/10.1109/ICSE-SEIP52600.2021.00043
13. Liu, P., et al.: Unsupervised detection of microservice trace anomalies through service-level deep Bayesian networks. In: ISSRE 2020, pp. 48–58. IEEE (2020). https://doi.org/10.1109/ISSRE5003.2020.00014
14. Ma, M., et al.: MS-rank: multi-metric and self-adaptive root cause diagnosis for microservice applications. In: ICWS 2019, pp. 60–67. IEEE (2019). https://doi.org/10.1109/ICWS.2019.00022
15. Ma, M., et al.: AutoMAP: diagnose your microservice-based web applications automatically. In: WWW 2020, pp. 246–258. ACM, New York (2020). https://doi.org/10.1145/3366423.3380111
16. Mariani, L., et al.: Localizing faults in cloud systems. In: ICST 2018, pp. 262–273. IEEE (2018). https://doi.org/10.1109/ICST.2018.00034
17. Meng, Y., et al.: Localizing failure root causes in a microservice through causality inference. In: IWQoS 2020, pp. 1–10. IEEE (2020). https://doi.org/10.1109/IWQoS49365.2020.9213058
18. Mi, H., et al.: Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Trans. Par. Dist. Sys. **24**(6), 1245–1255 (2013). https://doi.org/10.1109/TPDS.2013.21
19. Netflix: Chaos monkey. https://netflix.github.io/chaosmonkey/. Accessed 13 Aug 2022
20. Newman, S.: Building Microservices, 2 edn. O'Reilly Media, Sebastopol (2021)
21. Nguyen, H., et al.: PAL: propagation-aware anomaly localization for cloud hosted distributed applications. In: Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques. ACM (2011). https://doi.org/10.1145/2038633.2038634
22. Nguyen, H., et al.: FChain: toward black-box online fault localization for cloud systems. In: ICDCS 2013, pp. 21–30. IEEE (2013). https://doi.org/10.1109/ICDCS.2013.26
23. Qiu, J., et al.: A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. App. Sci. **10**(6) (2020). https://doi.org/10.3390/app10062166
24. Richardson, C.: Microservices Patterns, 1 edn. Manning Publications, Shelter Island (2018)
25. Samir, A., Pahl, C.: DLA: detecting and localizing anomalies in containerized microservice architectures using Markov models. In: FiCloud 2019, pp. 205–213. IEEE (2019). https://doi.org/10.1109/FiCloud.2019.00036
26. Shan, H., et al.: ϵ-diagnosis: unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In: WWW 2019, pp. 3215–3222. ACM (2019). https://doi.org/10.1145/3308558.3313653
27. Soldani, J., Brogi, A.: Automated generation of configurable cloud-native chaos testbeds. In: Adler, R., et al. (eds.) EDCC 2021. CCIS, vol. 1462, pp. 101–108. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86507-8_10
28. Soldani, J., Brogi, A.: Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: a survey. ACM Comput. Surv. **55**(3) (2022). https://doi.org/10.1145/3501297
29. Soldani, J., Montesano, G., Brogi, A.: What went wrong? Explaining cascading failures in microservice-based applications. In: Barzen, J. (ed.) SummerSOC 2021. CCIS, vol. 1429, pp. 133–153. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-87568-8_9

30. Thalheim, J., et al.: Sieve: actionable insights from monitored metrics in distributed systems. In: Middleware 2017, pp. 14–27. ACM (2017). https://doi.org/10.1145/3135974.3135977
31. Wang, L., et al.: Root-cause metric location for microservice systems via log anomaly detection. In: ICWS 2020, pp. 142–150. IEEE (2020). https://doi.org/10.1109/ICWS49710.2020.00026
32. Wang, P., et al.: CloudRanger: root cause identification for cloud native systems. In: CCGRID 2018, pp. 492–502. IEEE (2018). https://doi.org/10.1109/CCGRID.2018.00076
33. Waseem, M., et al.: Design, monitoring, and testing of microservices systems: the practitioners' perspective. J. Syst. Soft. **182**, 111061 (2021). https://doi.org/10.1016/j.jss.2021.111061
34. Weaveworks: Sock shop (2017). https://microservices-demo.github.io
35. Wu, L., et al.: MicroRCA: root cause localization of performance issues in microservices. In: NOMS 2020, pp. 1–9. IEEE (2020). https://doi.org/10.1109/NOMS47738.2020.9110353
36. Wu, L., Bogatinovski, J., Nedelkoski, S., Tordsson, J., Kao, O.: Performance diagnosis in cloud microservices using deep learning. In: Hacid, H., et al. (eds.) ICSOC 2020. LNCS, vol. 12632, pp. 85–96. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76352-7_13
37. Zhou, X., et al.: Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study. IEEE Trans. Soft. Eng. **47**(2), 243–260 (2021). https://doi.org/10.1109/TSE.2018.2887384