IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 30 November 2022, accepted 16 December 2022, date of publication 21 December 2022,
date of current version 27 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3231191

RESEARCH ARTICLE

A Study on the Aging and Fault Tolerance of Microservices in Kubernetes

JOSÉ FLORA, (Graduate Student Member, IEEE), PAULO GONÇALVES, MIGUEL TEIXEIRA, AND NUNO ANTUNES, (Member, IEEE)

Centre for Informatics and Systems of the University of Coimbra (CISUC), Department of Informatics Engineering, University of Coimbra, 3030-790 Coimbra, Portugal

Corresponding author: José Flora (jeflora@dei.uc.pt)

This work was supported in part by the Portuguese Foundation for Science and Technology (FCT) through the Ph.D. Grant under Grant 2020.05145.BD; in part by the Project Adaptive, Intelligent and Distributed Assurance Platform (AIDA), under Grant POCI-01-0247-FEDER-045907; in part by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation (COMPETE2020); in part by the Portuguese FCT under Carnegie Mellon University (CMU) Portugal; and in part by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020.

ABSTRACT Microservice-based applications are increasingly being adopted along with cloud service models, and nowadays serve millions of customers daily. They are supported by container-based architectures which are managed by orchestration platforms, such as Kubernetes, that monitor, manage, and automate most of the tasks. Although these tools provide failover capabilities, it is not yet studied how effective they are in dealing with diverse types of faults. Fault injection is an effective methodology for validating components that are supposed to detect the malfunctions and report/correct them. This paper studies the effectiveness of Kubernetes in dealing with faults and aging in microservices, and on the possibility of using faults to accelerate aging effects for testing purposes. For this, we conducted an analysis of the implementation and tuning of Kubernetes probes, followed by experiments with varying load and fault injection into two distinct and representative microservice testbeds to analyze the capacity of probes in detecting issues in applications. The goal is to improve the knowledge of researchers and developers on whether Kubernetes can detect different faults and aging issues. Also, even though some services tend to accumulate aging effects, with increasing resource consumption, Kubernetes does not detect them nor acts on them, indicating that probes may be insufficient for aging scenarios. Results also showed that fault injection is useful to accelerate aging effects for the testing and evaluation purposes.

INDEX TERMS Fault injection, fault tolerance, Kubernetes, microservices, software aging.

I. INTRODUCTION

The on-demand utilization of resources in cloud computing infrastructures is driving companies to adopt a microservice-based approach to devise and deploy applications [1], [2]. Lightweight software containers provide easy and fast instantiation contributing to efficient operations and adaptation to demand modifications. However, with large and complex applications, the number of microservices tends to become manually unmanageable. Orchestrators help with some of these tasks, by introducing automated monitoring and management to configure the deployment configurations. Orchestrators reduce costs and allow to put in place mechanisms that contribute to the resilience of the application, providing adaptiveness to different operation environments and even fault tolerance mechanisms. There are several orchestrators available for developers to use, such as Apache Mesos *(mesos.apache.org)*, Docker Swarm *(docs.docker.com/engine/swarm)*, or Kubernetes [3].

Kubernetes (K8s) is nowadays the *de facto* standard for microservices orchestration, adopted by many companies [4]. In terms of resilience, K8s provides three main features: automated horizontal scaling; restart of failing containers; and the ability to define probes to continuously assess the operation of containers [3]. Three types of K8s probes can be used to increase the resilience of microservice-based

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

systems [5]. The **startup probe** assures the application inside the container has started; the **readiness probe** indicates whether the container is ready to respond to requests; and the **liveness probe** indicates whether the container is running as expected. Still, the study of these resilience features has received limited attention.

It is not clear nowadays how effective are these features in detecting faults, namely faults related to software aging. In fact, aging is hard to detect, particularly in microservices, which have fewer resources available when compared to monoliths. For these systems, aging detection mechanisms require more effective and sensitive manners to swiftly detect and report the occurrence of aging. This would allow mitigation measures to act on the problem before the affected service becomes a bottleneck for the system. Also, the experiments conducted before deploying the services cannot be extended during large periods as in traditional systems, where their duration could take days or even months.

Previous work has shown that K8s probes are insufficient in the detection of microservices aging even though the resource usage of certain microservices grows extensively over the experimental period [6]. It focused on microservice aging, with experiments on a single e-commerce application demonstrating that aging should also be a concern in this domain. However, more software faults can affect microservice systems and can impair the normal operation of a microservice-based system.

This paper presents a study on the ability of Kubernetes in dealing with aging and faults in microservices and on the use of faults to accelerate aging testing. For this, we conduct a large experimental campaign in which K8s probes and two diverse widely used microservice architectures are analyzed in the presence and in the absence of faults. In a **first experiment focused in aging faults**, we study both architectures while under a regular operation profile, then while under a stressing load, and finally we inject faults aimed at emulating the accelerated effect of aging issues. The goal is to understand whether K8s can detect the aging issues and also to understand if injecting those issues can be useful for accelerating the validation of failure detection mechanisms. In a **second experiment**, both applications where exposed to several types of representative faults, to understand which types of faults the probes are able to detect. The injected faults were extracted from an industrial survey [7] and adapted to the applications under study.

To understand how to carefully create the K8s probes for microservices, we systematized the guidelines for their design and development, which are useful for developers to devise and leverage the checks performed during the operation of applications. This analysis reveals the necessity of considering the nature of the monitored service but also assuring that the probe is maintained, decoupled and concise. Then, the probes are used during experimentation. For each microservice application we followed a different approach. For `TeaStore` [8], we made use of the `registry` service, where every service of the application is registered when online, to perform the health checks. For `Sockshop` [9], we used the probes that were already configured, that use specific endpoints, in the deployment configurations available at the time on the public repository.

During the experiments, aging effects were observed in microservices that are highly requested, with the continuous increase of resource consumption, especially memory. However, K8s probes could not detect any of the aging-related failure. The results also show that the injected faults can be an effective way to test aging detecting mechanisms, as it allows to accelerate the effects of aging, resulting in less time required for aging experiments in the context of microservice-based systems. The experiments with a wider kind of faults also showed that K8s probes were not effective in detecting faults that affect microservices (no failure was detected, although several were observed) and thus, there is potential and need for work that improves these mechanisms. These findings are key for the decision-making process of developers and system administrators during the selection of the orchestrator for their applications, and also on how much to trust the probes they use.

The main contributions of this work are as follows:

- An experimental study on microservices aging, analyzing the ability that K8s probes have on detecting aging issues. The experiments included two diverse and widely used microservices applications, built using different technologies and architectures. The key insight is that K8s probes were not effective in detecting the observed aging failures. This is particularly significant considering the used probes followed two diverse and well established development strategies.
- An experimental evaluation that used fault injection to assess how K8s resilience features deal with failures in microservices. The study was conducted in two representative microservice applications, and the faults injected were obtained from an industrial survey on microservice faults, for representativeness. The key insight is that K8s probes fail to detect the occurrence of failures. This shows that probes require a deep understanding of the monitored services, and it still might not be enough to detect many types of failures.
- The illustration of the applicability of fault injection in facilitating the testing of aging detection and mitigation mechanisms for microservices, as it allows to accelerate aging effects, enabling a faster and simpler evaluation of these mechanisms in microservice environments.

The rest of the paper is structured as follows. Section II presents related work and details microservices dependability, with characteristics and challenges. Section III details the basics for the experimental studies. Section IV presents probes development best practices. Section V presents the aging results and Section VI presents results for the probe effectiveness for different kinds of faults. Section VII discusses the results and Section VIII provides threats to validity. Finally Section IX concludes.

II. BACKGROUND AND RELATED WORK

The resilience characteristics for microservices is a crucial mechanism to assure the management and orchestration of extremely complex systems that can have thousands of microservices. Dealing with different types of faults is a crucial topic on this context given the diversity of technologies that can be used to develop the microservices. It is very probable to have faults in large codebases, owing to time or budget constraints, thus it is utterly important to take them into consideration when building or deploying critical systems, using, for example, mechanisms that are able to tolerate faults.

A. SOFTWARE AGING

Software aging can be defined as the accumulation of resource utilization owing to faults present in the software released to production [10]. These faults, which are hard to detect during testing, have an effect on the system over time, tackling the performance and resulting in higher response times. As stated, faults are unavoidable, therefore, aging is likely to occur from the accumulation of faults' effects. Rejuvenation is the process of aging mitigation, or effect reduction, and, for instance, can be performed through restarting an application with the intent of returning the software to a clean state.

Software aging has been researched and several works are available, being applied to some real case scenarios [11]. Web servers were studied with different models used to monitor the system [12]. This work was then extended, focusing on Apache, concluding that when subjected to overwhelming workloads there are traces of aging but also minor rejuvenation [13].

However, aging in micoservices has not received significant attention. Previous work performs a preliminary study on the subject, applying known principles of aging to microservices and observing the results [14]. To detect aging, a deep learning model was applied and proved to be effective. Still, it does not elaborate on the topic, only proving that it is a relevant subject that needs more attention. Some past work has used the injection of faults to accelerate aging effects on several types of systems for testing and evaluation purposes [15], [16], but without focusing on microservices. Other works have focused on container engines [17] and containerized applications [18] revealing that aging is a concerning issue.

B. DEPENDABILITY AND MICROSERVICES

Fault tolerance has been studied on various subjects, especially on cloud computing, where three different surveys have been done [19], [20], [21]. Considering the most recent study, various different approaches are being used: system model, which is related to the network topology being used; proactive approaches, where the system acts preemptively to changes that may cause failures; reactive approaches, very similar to proactive, with the difference of reacting to the failures that already happened; and finally, we have some miscellaneous approaches that use machine learning and other techniques. In the end the authors suggest the study of deep learning or blockchain driven fault tolerance techniques, but also the challenge of having fault tolerance within systems that are using data deduplication and also an emphasis on performance issues [19]. Fault tolerance on microservices has also received considerable attention lately, with various studies on the subject [22], [23].

When it comes to Kubernetes, it has been a topic of research owing to increased use in industry, still mainly focusing on scalability. Several features are provided to address software problems in an autonomous manner, as for instance, the Horizontal Pod Autoscaler (HPA) that allows the autoscaling of services according to demand. Previous to scale up, it is common for the service to be under stress load for a short period of time, to better use the resources available and not frequently perform scaling up or down. HPA has been extensively researched and improvements have been proposed allowing developers for better results, with higher flexibility and effectiveness [24]. probes are an example of other approaches to monitor services and identify malfunction, which perform periodical health checks on the corresponding services [5].

Kubernetes has also received attention regarding fault tolerance, where, for example, it was proposed a multi-master platform tolerant to Byzantine and non-Byzantine faults [25]. Robustness in general is an known given by Kubernetes, due to many of its features being fault tolerance and assuring high availability [26]. Probes however, have not received much attention, although being used on many real life scenarios, therefore being one of the targets of our work [27].

Microservice-based systems are gaining ground in application design, development, and deployment, owing to their modularity [1]. Coupled with orchestrator management, they can leverage flexible and easy to use deployment. Kubernetes (K8s) is the most popular orchestrator on the market supporting diverse systems, even reaching business-critical systems [4]. Following, we detail some aspects of microservices and challenges raised in their environments.

Microservice architectures intend to overcome limitations of monolithic applications, composed of multiple tightly coupled components resulting in low modularity [1], [28]. Monoliths' advantage are the easy distribution and deployment, and inter-component communication. Problems arise as the monolith ages, since maintainability and dependency management become very hard [28]. Resource usage is not optimized, becoming cumbersome to effectively satisfy all deployment configuration requirements, leading to scalability limitations [28]. The idea of **microservices** is to decompose the different parts of the system into indivisible units of software that provide a well-defined function [1], [28]. A microservice can also be updated without a full application reboot, as required in a monolithic approach. As microservices are small, they are easier and faster to instantiate or remove. Thus, potentiating scalability and elasticity of the services and use resources as they are needed.

C. MICROSERVICES CHALLENGES

The use of microservices in production systems does not solve all known problems and, in some cases, it may even exacerbate them, creating difficulties in assuring satisfactory resilience levels for the applications [28]. For instance, the components that used to communicate through internal mechanisms are now decoupled and are required to expose an API to exchange information [28]. The network becomes more complex, with more data being exchanged, services need to assume trust in other components, that commonly use different technologies, and rely on them to achieve their goal. Also, the possibility of access from external sources or even from third parties deployed in the same infrastructure, potentially with different vendors, owing to cloud deployments and multi-tenancy environments cause the attack surface area to be greater than ever before [29]. Cloud environments used tend share physical resources with applications from multiple vendors, where some can obtain resources with malicious intentions [29], [30].

Microservices are easy to develop and test individually, but microservice-based systems is a different story. The interactions among services are complex and hard to cover completely. Some techniques, as chaos engineering emerged to test the systems in production [31]. Besides good development practices and use of reliability design patterns, orchestrators provide mechanisms to increase systems' resilience. K8s provides probes with the main goal of detecting failures in services and apply mitigation measures, such as service restart. Still, there is reduced work on the evaluation of how effective this mechanism is. It is important to understand which failures, and the nature of the responsible faults, K8s probes can detect in a microservice environment.

As reason for failures, the aging of software systems has been researched in the past along with its impacts on system performance [10], [11], [13]. Methods of rejuvenating the affected system, without significant impact availability, have been proposed [14]. However, most works have focused on the non-microservice approach [10], [12], [13]. Our previous work has addressed this issue and is now extended to consider other types of faults.

Even from a security perspective, missing a fault can potentially be maliciously leveraged to compromise the infrastructure [32]. Hence, this work also serves as a basis for deciding whether K8s features can be combined with security measures, such as intrusion detection mechanisms. The association of different mechanisms can significantly improve the resilience and security level of an application that is managed using K8s. Even if the probes cannot settle the issue entirely, this work can still contribute with a clarification about which set of faults are more prone to be detected resulting in possible scenarios where this mechanism can be applied.

III. EXPERIMENTAL CONTEXT

Based on the aging study conducted, we extend the work to evaluate the effectiveness of probes to detect and actuate upon failures resulting from different types of faults. Also, to summarize best practices for probe design and assess the careful tuning of the parameters available. The practical experiments conducted in this work intend to study the Kubernetes' ability to increase the resilience of microservices.

In this section, we describe the context that is common to the experiments described in Section V and Section VI. These experiments were performed on two different representative microservice applications, with different workloads, while collecting metrics with multiple tools.

A. REPRESENTATIVE MICROSERVICE APPLICATIONS

In this work, we selected two different and representative microservice applications, `TeaStore` [8] and `Sockshop` [9]. Both applications were developed with the intent to be used in practical experimentation with microservices. Fig. 1 and Fig. 2 depict the architecture of TeaStore and SockShop, respectively, that are detailed next.

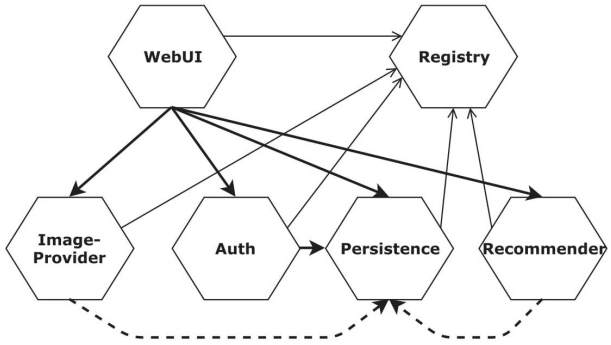

FIGURE 1. `TeaStore` **architecture (from [8]).**

`TeaStore` is composed of `5` java-based services designed to discover themselves through a `Registry` component. Each running service instance announces itself to the `Registry` as a way for the application to operate regardless of the active service replicas and their distribution. The `WebUI` service takes advantage of the `Recommender` service to provide recommendations to users. The system is also composed of a database that stores all the relevant data

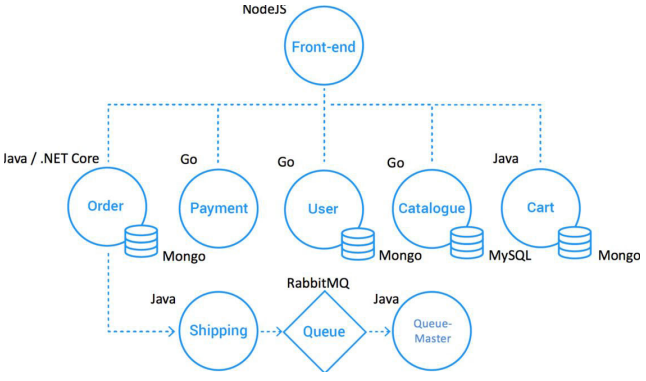

FIGURE 2. `SockShop` **architecture (from [9]).**

and is only accessed through the `Persistence` service. All services communicate via REST, as this is established as the de-facto standard in the micro-service domain [8].

`SockShop` simulates a user-facing e-commerce website that sells socks that are intended to aid in the testing and analysis of microservices and cloud-native technologies [9]. The application is composed of `8` services, developed in different programming languages, such as NodeJS, Java, and Go, and four databases. Similar to TeaStore, SockShop also uses REST for the communication between services.

To deploy the applications, we followed the guidelines available at the applications' GitHub repositories and deployed each into a Kubernetes cluster. We performed preliminary experiments to tune the workloads intensity and the applicability of the user behavior profiles. These tests had the main objective of assuring the services were not stressed outside the stress scenario and faulty systems were operating effectively. During the final experiments, we consider two replicas for the `WebUI` and `Persistence` services as during preliminary testing these were subjected to most issues related to scalability concerns.

B. WORKLOAD

To exercise the microservice testbeds, we utilized the Locust *(locust.io)* load testing framework. Locust is an easy-to-use, scalable tool that allows defining the behavior of users that interact with the target application.

For `TeaStore`, we created the clients' behavior profiles based on the request profiles available at the public GitHub repository; there are two types of interactions: *browse*, that emulates users browsing the store, selecting items and adding them to the shopping cart as if they would purchase them, but no orders are submitted; *buy*, emulates users browsing the store and conducting purchases of selected items. The *buy* profile was used during the experiments, as it is a more complex and representative workload.

For the `SockShop` application, the interaction profile publicly available was already ready to be used with Locust. In this profile each transaction includes the user connecting to the application, listing the items available, and complete purchases.

C. MEASUREMENTS

To understand the performance and behavior of each system multiple tools were used for different purposes. Information about the transactions executed using Locust, Kubernetes Metrics *(github.com/kubernetes-sigs/metrics-server)* to collect CPU and memory usage per service, and Kubernetes Events to extract information about the status of the pods. The measures for CPU and memory are the ones typically used in Kubernetes. For CPU, we use the millicpu that corresponds to one-thousandth of a CPU unit. One CPU unit is equivalent to one physical CPU core, or one virtual core [33]. The memory is measured in bytes.

The transactions performed allow us to keep track of the system's ability to respond and the impacts that fault, or more specifically, aging, have on the system. The use of Kubernetes Metrics is required to observe the effects of load changes and faults impact in the resource usage of the system, allowing to verify whether the impacts are noticeable. Events are a resource type in Kubernetes that are created when components have errors, changes in state, or other messages, and are useful to debug malfunctions in the system [34]. Because containers have a status that can be modified, we can use Kubernetes Events to log information regarding whether a probe check detected an erroneous state in its operation. If so, the container is *unhealthy*, otherwise it is *healthy*. Using the typical commands of *kubectl* to get information on the active pods and resources, the logs of each service and pod, and the detailed information from the describe command was enough to understand the state of the container.

D. EXPERIMENTAL SETUP

The experimental setup consists of three machines. The Kubernetes master machine has a processor with 2 cores and 16GB of RAM, while the worker node machine runs with 4 cores and 16GB of RAM. The Test Driver runs on a separate machine to generate the workload using 2 cores and 16GB of RAM. All three machines are running Ubuntu and operate as depicted in Fig. 3, with the Kubernetes cluster being composed by a master and a worker.

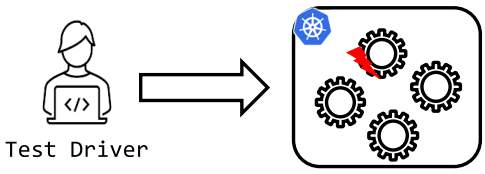

FIGURE 3. Experimental setup: microservice application deployed in Kubernetes, with a fault injected in a microservice.

The Test Driver controls the execution of the experiments, emulating the clients that are using the system, and for the activation of the faults. It includes the workload generator based on Locust that executes the workloads described previously. The Kubernetes Cluster deployed for each testbed, following the recommendations available, used the two machines available and monitored the resource usage of each active service and continuous probe operation on the lookout for possible malfunctions.

IV. KUBERNETES PROBES

As a part of its adaptive capabilities, Kubernetes (K8s) provides three different probes whose objective is to assure that applications are delivering the expected service and ready to respond to incoming requests [5]. These mechanisms have the potential to be utilized in a wide range of critical applications, that are nowadays deployed in Kubernetes. There is an urge to understand probes and analyze their advantages and drawbacks.

Our goal is to analyze and understand the relevant features in the design and implementation of K8s probes for

microservice applications. Comprehending the design decisions made for the monitoring of the service probed and the results obtained so these mechanisms are indeed contributing to system-wide fault tolerance. This knowledge can contribute to more informed decisions during the design procedure.

Of the probes in Kubernetes, the most relevant are [5]:

- **Readiness probe:** assesses whether a container is ready to take requests. In the event of failure, the endpoint controller removes the IP of the pod containing the failing container from all the endpoints that match it. A pod is considered ready when all its containers are ready. Otherwise, it is removed from the service load balancers.
- **Liveness probe:** consistently conducts the probing of the container to assess its running state. If a failure is detected the container is killed, and the restart policy controls the next operation performed. When a container is killed, a new one is created.

A probe can be defined using three different approaches: an HTTP request (e.g. a GET request); a TCP connection, where a connection is established to a specified port; and a user-defined command (e.g. check if a file exists).

After the container starts, the startup probe validates the correct initialization of the container. Then, the readiness and liveness probes kick in. The readiness probe checks if the container can receive and process requests. The liveness probe continuously performs checks and restarts the container when it detects a failing state.

A. BEST PRACTICES FOR PROBE DESIGN

The design and implementation of probes is a delicate procedure that will have an impact on detection effectiveness. It can also impact the performance of a specific service and have repercussions on the whole system. The definition of the readiness and liveness probes needs to take thoughtful consideration to maximize the outcome from its utilization. So, they provide the system with effective resilience and self-adaptiveness levels.

Based upon prior knowledge and analysis from experience [5], [35], a set of best practices emerge. They are presented so that K8s probes are more effectively designed and implemented:

BP1 Independent application handlers: create specific handlers in the application for processing the requests from each probe. This allows each probe to have a specific endpoint to call on and remain an integral part of the application, which is important due to K8s's additional monitoring mechanisms. Abiding by these recommendations allows developers to have an assurance that probes will not cause complications in the normal operation of the application.

BP2 Focused goals: the handlers designed for each probe should conduct only verifications and nothing else. For the readiness probe, the required components for the successful operation of the service should be checked. For the liveness probe, the status of operation should be verified and returned as a simple code (e.g. `OK` or `NOT_OK`).

BP3 Report without correcting: probes must not attempt to correct any errors that may be identified. The only objective is to understand whether the system is operational. Correcting the problems identified is the responsibility of other mechanisms.

BP4 Parameters tuning: Besides the correct definition of the probes, tuning the parameters is important. Careful tuning of the parameters' configuration should be done so that it does not become an obstacle to the service's operation.

The most relevant parameters are the `periodSeconds`: which specifies how often to execute the probe; the `initialDelaySeconds`: which specifies how long to wait to initiate the probe after the container starts; and the `timeoutSeconds`: which specifies the number of seconds after which the probe is considered failed [5].

A small value for the `periodSeconds` parameter could force the probe to execute too many checks. In contrast, a large value could leave the container in a broken state longer than desired. When the value of the `initialDelaySeconds` parameter is too small, the probe might start the diagnosis before the application is ready, and it will force a wrong restart of the container or the pod to not accept traffic. This requires the programmer to leave time to start the application inside the container. Regarding the `timeoutSeconds` parameter, a short timeout may lead to false alarms because the application did not have enough time to process the action, whereas a large timeout might leave a broken container running longer than expected.

B. ANALYSIS OF PROBE DESIGN AND IMPLEMENTATION

In this paper, we use two different approaches for K8s probes implementation. For `TeaStore`, we made use of the `registry` service, where every service of the application is registered when online, to perform the health checks. For `SockShop`, we used the probes that were already configured in the deployment configurations available at the public repository. Each `SockShop` service exposes an endpoint that returns the health status of the service. These endpoints are used by HTTP liveness and readiness probes.

We then conducted a qualitative analysis of the implementation of probes for `TeaStore` and the difficulties that may emerge. In our experiments, we aimed at providing each of the five services with a liveness probe. Hence, we studied the configurations and actuation mode of the microservices in the application to understand which were the important points where the probes should focus. Primarily, the communication between services is performed using REST. Moreover, the registry previously mentioned congregates the list and corresponding information of each active service. The infrastructure utilizes a heartbeat mechanism that assures the

correct and continuous availability of each service since when the heartbeat check fails, the service is removed from the list of active components.

Regarding the implementation, we started with an analysis of the system to understand it. We used a request to the status page of the `WebUI` service to understand if it was operating correctly (see Listing 1). However, this would cause significant overhead to the service and this method satisfied only two of the four best practices proposed, **BP2** and **BP3**.

```
livenessProbe:
    exec:
        command: [
            "wget",
            "http://<service_ip>:8080/tools.
    descartes.teastore.webui/status"
        ]
    initialDelaySeconds: 120
    periodSeconds: 10
    failureThreshold: 3
```

Listing 1. Initial (naïve) liveness probe for the WebUI service.

Further versions of the probes leveraged the heartbeat checks of the `registry` service. So, the probe was modified to be focused (satisfying **BP2** and **BP3**) and use an independent handler, which satisfies **BP1** (see Listing 2).

```
@GET
@Path("{name}/{location}")
```

Listing 2. Endpoint created for probes.

So, to satisfy **BP4** the period of the checks was decreased from `10` to `5` seconds and the remaining parameters tuned, see Listing 3.

```
livenessProbe:
    exec:
        command: [
            "/bin/bash",
            "-c",
            "curl -f http://<registry_ip>:8080/
    tools.descartes.teastore.registry/rest/
    services/tools.descartes.teastore.persistence
    /<service_hostname>:8080"
        ]
    initialDelaySeconds: 120
    periodSeconds: 5
    failureThreshold: 3
```

Listing 3. Liveness probe created.

The configuration changes were demonstrated to have an impact on probe operation. As there are multiple probing moments, the requests should be lightweight and direct to the goal. Clearly, this was not the case in the first implementation attempt, as the probe received a complete HTML page where a simple status code was enough. A more thoughtful method was devised and allowed the probe to become faster and reduce the overhead caused in the service. These insights allow developers to have a better understanding of the conditions and steps required to devise and implement K8s probes for microservice applications. It can be useful when deciding whether probes should be implemented. It also helps to guide their implementation, focusing on some aspects that require special attention.

Regarding the probes already implemented in SockShop, these are specific endpoints for each service that report the ability of the service to operate without correcting any problem found with it. Hence, the probes already present in SockShop satisfie the best practices referred previously.

V. MICROSERVICES AGING EXPERIMENTS

Kubernetes (K8s) supports diverse microservice applications, as business-critical systems, that require different levels of resilience and performance. K8s provides different features, such as probes, that allow increasing the resilience of systems. Still, there is little information regarding their effectiveness, specifically in the identification of service failures when aging affects the microservice. In this paper, we present an extension of the work previously published [6], to a different testbed so that results are more consolidated.

The objective of this experimental study is two-fold: i) evaluate the detection effectiveness of Kubernetes probes of aging effects in microservices; and ii) study the acceleration of aging effects in microservices through the utilization of software faults.

For this, we devised an experimental campaign based on representative microservice-based systems and applied three different scenarios. These scenarios consist of a normal operation, a stress operation, and the utilization of an aging-related fault in a stress environment. The information obtained from this study contributes to a clearer picture of the protection granted by K8s probes. Further, it also provides the possibility of understanding whether aging can be accelerated to usable timeframes within the typical constraints of the development and deployment of microservice-based systems.

A. EXPERIMENTAL DETAILS

Building upon the experimental context described in Section III, we defined the following specific details for the aging experiments. During this work we devise different operation environment scenarios (detailed below) and analyze the behavior of the services in each one. The main goal is to look for aging effects and understand how K8s probes react to them. Based on the results obatained over the experiments with the stress load, we injected faults in the `WebUI` service of `TeaStore` and the `orders` service of `SockShop` beacuse these were the services most affected by the stress scenario. For the experiments we only injected a memory leak fault that is active throughout the duration of the experiments and tends to accumulate memory resources. The fault used meets the requirements of a typical fault injection procedure for the purpose of causing aging in the software components it affects [10], [13], [14].

To analyze the aspects of microservice aging, we devised three experimental scenarios for the operating environment of the system. The **simple scenario** consisted of a normal

operation of the system over 48h exercised with the normal workload depicted in Fig. 4, having 9 active users over time. The **stress scenario** executed over a period of 48h with the use of a stress load that is represented in Fig. 4, therefore overloading the system to provoke aging behavior under stressful conditions. The **faulty scenario** is a merge of the stress scenario with the presence of a fault in the WebUI and orders services of the applications. This scenario runs for a shorter period, only 24h, as it is enough to demonstrate the intended use of accelerating the effects of aging in microservices.

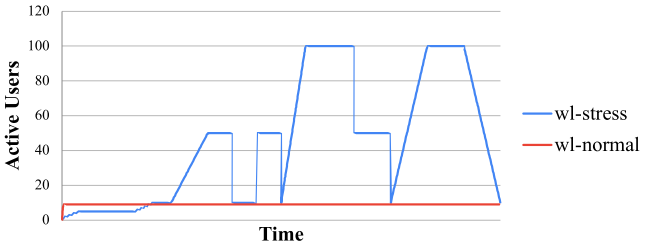

FIGURE 4. The load intensities used for the two workloads.

Each scenario intended to accomplish the following goals, respectively: i) have the baseline behavior for the system; ii) obtain the effects of aging for the system under stressful conditions and iii) understand whether faults can be used to accelerate aging effects in microservices.

B. RESULTS FOR AGING EXPERIMENTS

The results obtained during the experimental campaigns with the TeaStore and SockShop applications are presented in Fig. 5.

For TeaStore, services Recommender and Auth and DB were suppressed as their behavior was similar to Registry and Image services, respectively, thus not adding much and allowing the visualization to be clearer. For similar reasons, we also suppressed information about some services of the SockShop application. Namely: catalogue, session-db, shipping, user, and user-db services are not depicted.

In both figures, the first row contain information related to CPU usage, the second row presents data from memory consumption, and the third row presents data about the transactions executed by the system. The experimental scenarios are presented from left to right: simple scenario, stress scenario, and faulty scenario.

Across all scenarios, the systems are exercised with the corresponding workloads, as described in Section III. It is possible to observe the consumption of resources that denotes unsteadiness at the beginning and the end of the experiments, consistent with the warm-up and cool-down phases. These phases are not considered in result analysis, focusing only on the system's steady phase. During this period, microservices are warmed-up and functioning as normally would in a real environment.

1) CPU USAGE ANALYSIS

For TeaStore the CPU utilization is very stable for each scenario analyzed, although some spikes are observed. For SockShop, there is a similar behavior with the exception that during the faulty scenario, the orders service shows two spikes in usage relative to crashes experienced by the service. However, among the three scenarios there are clear differences in the usage of CPU for both testbeds.

For TeaStore, the simple scenario shows a quite stable CPU consumption for all services, with an upper bound around 200 millicpu. For SockShop, there is a stable consumption across every service during the stable scenario. The general CPU utilization remains at low levels, without crossing the 200 millicpu line for any service. This shows that, when the load is manageable, the services have very steady and low usage of CPU.

In the stress scenario, the spikes in some TeaStore services (mainly Persistence and WebUI) are constantly increasing, crossing 400 millicpu by the 16^{th} hour of the experiment and 600 millicpu by the 30^{th}. For SockShop, there is a noticeable oscillation in CPU consumption for every service. This indicates that resources are used according to the level of user demand. There is a set of four crashes for the front-end service around the 26^{th} hour of the experiment, which corresponds to a identifiable spike in CPU usage in the Figure.

On the faulty scenario, there are a few oscillations in TeaStore, similar to the simple scenario, with the CPU consumption hardly crossing the 400 millicpu value. Still, in the second half, there is a slight increase of oscillations to higher levels, as in the stress scenario. For SockShop, the behavior of the services remains quite similar to the observed during the stress scenario. However, there are two moments in the experiments where the CPU usage of the orders service spikes, reaching around 900 and 1300 millicpu. These moments correspond to crashes that orders experienced during the faulty scenario, around the 9^{th} and 18^{th} hours. The patterns of CPU usage for both cases is quite similar, with the second crash's spike being slightly more elevated.

Overall, the CPU consumption is affected by the stress load that is directed to the system, while it remains relatively stable and with lower spikes over the simple scenario in both testbeds. However, SockShop orders service experiences two crashes during the experiment period.

2) MEMORY USAGE ANALYSIS

The observations of memory usage depict more dissimilarity between services in TeaStore than in SockShop. Some TeaStore services, such as WebUI and Persistence, suffer from the accumulation of memory over the period of the experimental campaign. Yet, the services have different degrees of accumulation, as WebUI grows faster than Persistence. SockShop services carts-db and orders-db demonstrate a pattern of continuous increase of memory use.

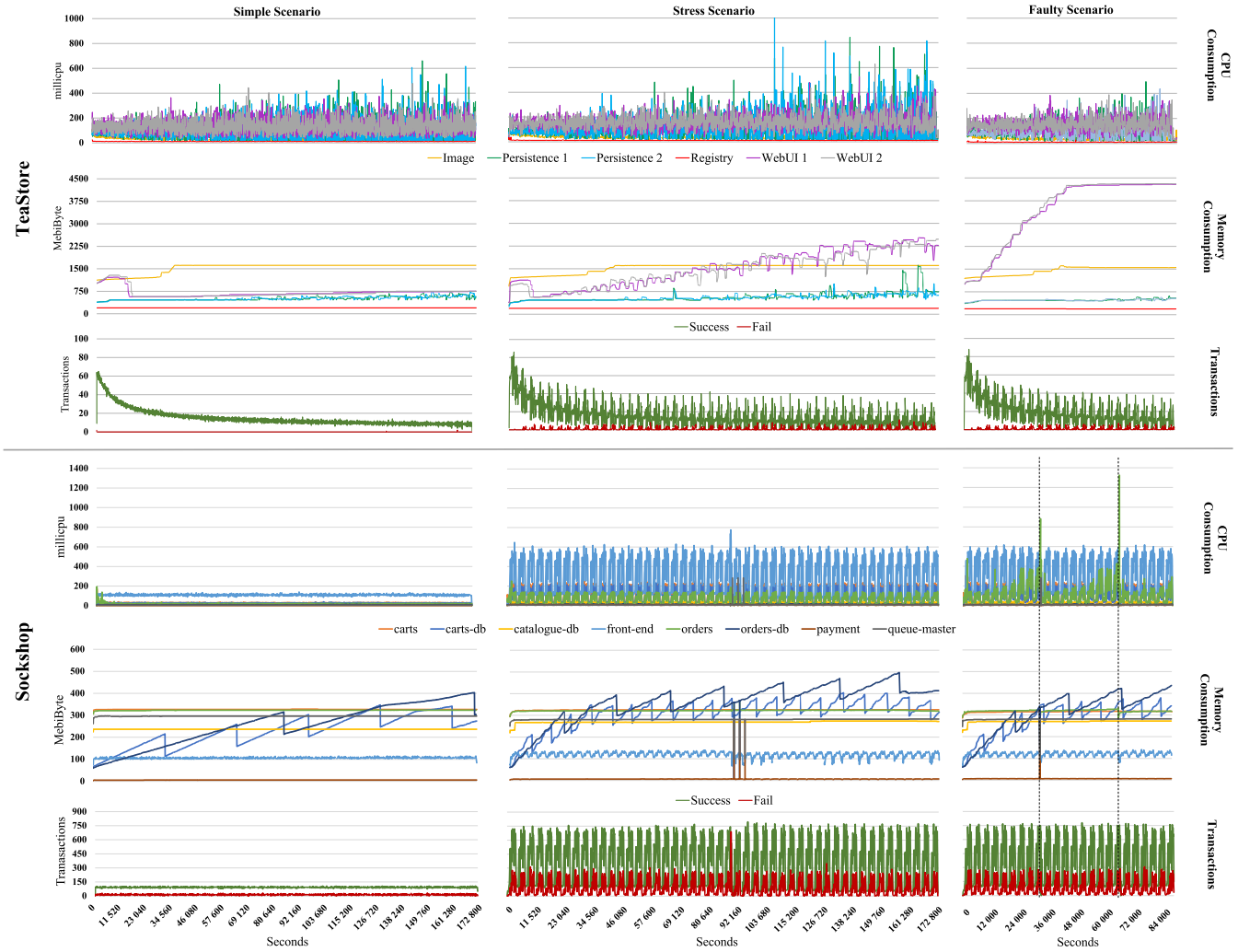

FIGURE 5. `TeaStore` and `Sockshop` results with the memory, CPU consumption, and transactions for each scenario. Top corresponds to CPU, middle to memory and bottom to successful and failed transactions. From the left to the right, we have: simple scenario; stress scenario; and faulty scenario.

The analysis of the simple scenario shows very consistent memory usage across all the services. `TeaStore` `Registry` and `Image` services remain very stable, only showing slight increases over the experimentation period. On the other hand, `WebUI` and `Persistence` have a trend to accumulate memory. This indicates that these services are more prone to suffer from memory accumulation and aging effects. For `SockShop`, all but the `carts-db` and `orders-db` services remain constant with their consumption of memory. These database services are based on MongoDB and can be having issues managing the connections established.

During the stress scenario, we observe similar behavior for all `TeaStore` services except for `WebUI` and `Persistence`. When subjected to stress, `WebUI` service instances grow their memory consumption from around `300%` (from `600` to close to `2500` MebiBytes). This is a concerning factor when dealing with microservices that tend to be under stress for a while before scaling events take place. `Persistence` has a smaller increase. `SockShop` `orders-db` and `carts-db` have increasing memory consumption. This may indicate that stressed services could suffer from aging if their workload varies frequently. For these reasons we decided to instrument the fault in the `TeaStore` `WebUI` and `SockShop` `orders` (which interacts with `orders-db`) services.

The faulty scenario has a shorter experimentation period. Still, we successfully achieve more intense aging effects on microservices. The fault injected in the `WebUI` service increases the memory usage by more than `600%` (from `600` to close to `4300` MebiBytes) in each replica. This is accomplished in just under `10h` of experimentation. For `SockShop`, the `orders` service cannot continue to process requests as it becomes too caught up the fault injected in the service and crashes two times.

TABLE 1. Faultload used in the experiments: faults based on the results presented in [7] and classified according to [36], with the services where each fault was injected on both testbeds.

ID	Description	Persistence	Reproducibility	System Boundaries	Root Cause	Influence	Frequency	TeaStore	Sockshop
F3	The system periodically returns server 500 error	Transient	Mandelbugs	Internal	Environment	Non-Functional	High	Recommender	orders
F4	The response time for some requests is very long	Transient	Mandelbugs	Internal	Environment	Non-Functional	High	Recommender	orders
F5	A service sometimes returns timeout exceptions for user requests	Transient	Bohrbugs	Internal	Interaction	Non-Functional	Low	-	-
F7	Error in the business logic	Transient	Bohrbugs	Internal	Interaction	Functional	Medium	WebUI	front-end
F15	Failure of the database	Persistent	Bohrbugs	External	Environment	Functional	Low	teastore-db	catalogue-db
F15t	Intermittent failure of the database	Transient	Bohrbugs	External	Environment	Functional	Low	teastore-db	orders-db

Therefore, the use of faults can significantly reduce the time required to achieve higher levels of aging effects demonstration, which in turn makes the evaluation of mechanisms and tools that should detect aging on microservices more practical.

3) TRANSACTIONS ANALYSIS AND K8s PROBES DETECTION

`TeaStore` successfully responds to most of the transactions it receives over the three scenarios under study. On average, it fails to respond to 0% during the simple scenario; it has a 6% failure rate during the stress scenario, and 5% during the faulty scenario. Although, the failed transactions rate is smaller during the faulty scenario, the experiment only runs for `24h` and demonstrates an increasing pattern. The successful transactions vary according to the workload intensity used and it is possible to see the system tending to respond to fewer transactions successfully over time. This observation is a consequence of resource accumulation. In `SockShop` there is a clear increase of transactions satisfied with the stress scenario. This is also true in the faulty scenario. For the simple scenario, `SockShop` has a 13% failed transactions rate while during the stress and faulty scenarios it increases to 17%.

Although there is several service restarts for `SockShop`, none is because of probes. The K8s default mechanism restarts the front-end service four times, all around 23^{th} hour, during the stress scenario; and it restarts the orders service two times, around the 9^{th} and 18^{th} hours of experimentation of the faulty scenario. The K8s probes mechanism for aging effects detection on microservices fell short, not detecting any problem with the services.

Overall, the main observations are that aging also affects microservices and that Kubernetes probes are not able to detect aging issues and thus do not act effectively on their correction through service rejuvenation. The experiments demonstrate that a stress load can cause resource accumulation which can be accelerated through fault injection so that it is possible to have faster effects.

VI. KUBERNETES FAULT TOLERANCE EVALUATION

Following the aging experiments conducted, it was clear that Kubernetes (K8s) probes were unable to detect aging in microservices. So, we widened the study to understand whether this is also valid for different types of faults that also affect microservices.

The goal of this experiment is **focused on evaluating the effectiveness of the probes** in terms of failure detection rate capabilities, and understanding which type of faults are more prone to result in probe actuation. This information is greatly valued by developers whose responsibility is to assure the system's maximum availability and continuous operation. For this, we devised a complete fault injection procedure executed through an experimental campaign that is further detailed to bridge this gap and provide empirical knowledge. The experimental campaign is based on two representatives testbeds where faults were injected in different services and activated during the monitoring of the system.

For `TeaStore`, the required probes were designed and implemented following the best practices elicited in Section IV; for the case of `SockShop`, the services already had probes implemented. Further, we present experimental details and the results obtained during the experiments.

A. EXPERIMENTAL DETAILS

The experimental campaign followed a three-step methodology. Initially, we set up the infrastructure, deploying the corresponding application into K8s, starting the load generator after a waiting period for the initialization of the application. At the midpoint of the experimental slot (`15min` after starting), the fault is injected, and the slot remains operating for other `15min`, resulting in a `30min` slot period. At last, the information is collected from the K8s features and analyzed the output to identify whether the probes can detect the malfunctioning of the system. To serve as a baseline of the service behavior, we also execute a series of **golden runs**, which are executions without the injection of a fault.

Fig. 6 presents an overview of the experiments' methodology with the slots performed. Each fault was injected in three different slots for each workload used. For each repetition, the seed feeding the random generator for the selection of requests issued was modified accordingly so that we assure the independence between executions. For the experimental runs, we devised and implemented an intensity load

profile characterized by an oscillating pattern of active clients between 4 and 35 with a period of 10min between peaks.

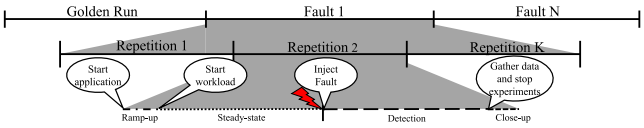

FIGURE 6. Overview of the fault injection methodology.

B. FAULTLOAD

The faults presented in this work are based on an industrial survey that condenses the typical faults observed in microservice systems and the difficulties developers face [7]. The work then replicates the identified faults into a representative microservice benchmark and analyze the current debugging practices and study possible improvements. Based on the set of faults identified in the work, we selected the ones that are representative of different categories and most applicable to the context under analysis in this paper, these faults are generalizable, and they can be found in any application that follows this architectural design. Parts of the deployment and the source code of the applications were modified to include the faults presented in Table 1. These faults are now part of the testbeds used in this work and can be activated as needed during runtime through a dedicated endpoint. The faults elicited have the potential to cause different failure modes in the microservices they affect. Based on the CRASH scale [37] the different faults elicited, can cause every one of the states on the microservices they affect. However, probes are only expected to identify cases where either the catastrophic, restart, or abort failure modes are observed. These are the ones where the system has delivery problems that can cause the service to fail the health check performed by the probe system.

As we can see in Table 1, we have six faults covering different defects in a microservice application. We represent periodic failures with internal server errors (fault F3); Server response delays through fault F4; fault F5 at times returns exceptions to user requests; fault F7 covers a business logic error in the application; and, F15 and F15t represent a failure of the database, a required core part of any system. All the faults are focused on the operation of the service that they affect and do not consider specific characteristics of each service as the transactions or requests processed, as this would decrease the representativeness of the fault. In this part of the work, we do not analyze aging-related bugs as that was the focus of Section V and was extensively analyzed in there.

As we can see on the right of Table 1, TeaStore faults were injected in Recommender, WebUI services, and the database. On Sockshop the injection happened in the front-end and the orders services, and the databases for the orders and the catalogue services.

Most of the faults are transient, however, there is one fault (F15) that is persistent. As all faults are software faults, we classified them according to the reproducibility of the bug: as Mandelbugs when there is difficulty in its reproduction and detection (F3 and F4); and Bohrbugs for deterministic failures that are easier to spot (F5, F7, F15, and F15t). All the faults occur during the operational phase and are perceived by the user when there is a failure.

In microservices environments, faults are complex and can propagate across services. For instance, the fault where the database is down for a while can cause failures to manifest in other services although not being its root cause.

Some characteristics are inherent to the nature of the faults injected. Namely, the influence caused in the system where they occur. Functional faults cause malfunctions of the system, as error manifestation or unexpected outputs. Non-functional faults degrade the quality of service, as performance penalties or increase the probability of the system becoming unavailable [7].

From the root cause perspective, the faults presented have two causes: environment, where the typical cause of the malfunctions lies on runtime execution environment configurations; and interaction, resulting from the interactions among the multiple services and customers [7].

The frequency of a fault provides information related to the number of times it is injected over the experimental period, which represents its real-world frequency. These faults can be of low, medium, or high frequency.

C. RESULTS

After the experimental campaign, we collected the data generated. This data allows us to understand the capacity of the probes in detecting failures that originated from the faults. As described, the experiments executed are composed of six different faults that affect different services in the applications. All services are instrumented with probes to detect malfunctions that, when detected, would lead to the application of the restart of the component. The results obtained are presented in Table 2. The numbers presented are the results of the average of several experiments slots, as we repeated each execution three times. Following we analyze the results produced in connection with the different characterization of the faults activated. We start with a general analysis disregarding the specific categories or nature of the faults and then perform a more focused discussion.

All the faults injected were correctly activated, resulting in errors and then in failures [36]. Still, none of the failures is detected by the probes monitoring the services. In either case, TeaStore or SockShop, the failures result in the K8s probes raising any alarm or performing any restarting of the service.

Regardless of the bug reproducibility inherent to the occurring fault, Mandelbug or Bohrbug, the probes were unable to pick up failures in the services of both systems. The boundary definition was not also sufficient to cause probes to react to failures. It was expected that internal failures were detected with more ease, however, both external and internal faults were not picked up. Similar behavior was verified for the root cause and influence of the faults.

TABLE 2. Kubernetes probes failure detection effectiveness.

Fault ID	# Faults Activated		# Failures Observed		# Failures Detected		% Failures Detected	
	TeaStore	SockShop	TeaStore	SockShop	TeaStore	SockShop	TeaStore	SockShop
Golden	0	0	0	0	0	0	-	-
F3	15	15	3	3	0	0	0%	0%
F4	15	15	3	3	0	0	0%	0%
F5	1	1	**100+**	**100+**	0	0	0%	0%
F7	15	15	3	3	0	0	0%	0%
F15	1	1	1	1	0	0	0%	0%
F15t	3	3	3	3	0	0	0%	0%

The probes did not detect any of the faults in the database (F15 and F15t). These faults cause a failure in a service that had not a probe configured. Nevertheless, the failure propagated to the operation of the service that interacts with the database. It causes the service to be unable to process and respond to requests. As a result, the failure should be noticed by the probe monitoring the database API service (e.g., Persistence). This indicates that probes are checking the liveness/readiness of the services but are not capable to perceive failures in a dependent service. Also, these faults diverge in their persistence, with F15 being persistent, and (F15t transient, which indicates that higher frequency may also be insufficient to assure detection of failures.

Despite no detection from probes, K8s restarts the pod of the database service when it is affected by fault F15t. The orchestrator can detect that the container is having issues owing to the intermittent failures it is experiencing and restarts the pod as a manner to attempt at solving the malfunction. There is no attempt at restarting the pod for the case of fault F15. In both TeaStore and SockShop the behavior of K8s is the same. This mechanism takes place because of the restart policy defined for the service and K8s detects that the container is in a failing state.

VII. DISCUSSION

Aging effects can be observed in microservices as in traditional monolithic applications, (cf., Fig 5). Monolithic applications have their complexity exaggerated when compared to microservices, however, these services tend to be stressed a reasonably higher number of times due to the application of auto-scaling mechanisms employed. Although this provides good quality attributes regarding scalability and elasticity, the use of replication mechanisms also leads to more fault accumulation resulting from the stress periods. The longer the auto-scaling mechanisms need to actuate, the worst aging effects will be experienced by the microservice, as stress is not reduced in time, which can impair their normal operation.

The utilization of faults in microservices can reduce the time required to observe the aging effects. Considering the scenario described with aging fault injection, even though it was executed for 24h, the effects seen are more damaging than the ones experienced during the stress scenario, (cf., Fig 5). Therefore, faults can accelerate testing processes before deployment, resulting in a reduced risk of affecting the performance, and even security, of the complete system. Research being conducted on the aging of microservices can also take advantage of this to make faster experiences. The fault used can be configured to further accelerate or even decelerate the aging process, allowing the adjustment of the microservice resource consumption. It is worth noting that a single slow microservice can deem the whole system dead, by disabling it from performing any requests or transactions, if the system is affected by the slowness is critical and the bottleneck to the whole system. Aging faults can be used to identify bottlenecks of the system, or even to evaluate microservice aging detection techniques, methods, or mechanisms, all promptly.

Kubernetes mechanisms were unable to detect both the aging and the faults of the microservices affected. In our experiments, K8s probes did not detect any failure in any of the microservice applications, (cf., Table 2). The failures perceived were not significant for the sensibility of the mechanism. It seems to only detect significant faults that cause the service to completely stop responding or have a noticeable hanged state. However, the use of probes in K8s clusters remains advisable. Still, developers should be aware of the limitations of the technique even when the best practices are applied correctly. The use of an approach, such as the one presented here, has flaws and constraints in detecting faults across diverse types and frequencies, although being able to contribute with some assurances. The criticality level of the system must be weighed in when deciding which self-healing mechanism should be applied.

VIII. THREATS TO VALIDITY

The experiments performed allowed us to perform a comprehensive analysis on the resilience of Kubernetes, evaluating its ability to detect both aging and faults with the use of probes. It provides both developers and researchers various findings that can be useful when dealing with these probes, allowing them to better understand their limitations and their use. However, it is still worth noting some threats that may rise and create doubts towards the validity of the work performed.

Regarding **internal validity** no real threats can be raised, since every experiment was done independently and various metrics were used. As for **external validity**, **population validity** may be considered a problem due to sampling bias, which, in our case, is also related to content validity, as in:

Only one configuration of the probes was used. Probes can be configured in order to adjust reaction times, being also directly correlated to their sensitivity. If the probes are too sensitive, it may say trigger a pod to restart even if it was just experiencing a momentary bottleneck, however the reaction will be really fast. We decided to only use one configuration in order to simplify the problem and the experiments. Nevertheless, we followed probes best practices, which are representative of a real case implementation.

For the experiments regarding fault tolerance, only 6 different faults were used. Faults have many classifications according to different characteristics. The faults we choose, although few, are representative of various categories. Therefore it can be argued that they are representative of a reasonable variability of scenarios, being enough to evaluate probes' robustness accurately.

It can also be a threat, when considering the experimenter effect:

Aging experiments were executed only once, therefore random factors may have affected the experiments, and consequently, the results. Given the nature of the experiments, randomness is prone to happen due to network latency and even task scheduling, however, we feel like these differences are negligible when considering the whole runs, with these being extended periods of time (`24h` and `48h`). Therefore, even if we had done more repetitions, we believe that the results would not vary much, and the research insights found would stay the same.

In the end, we think that these are not enough to nullify the findings made, which are important research progresses that should be taken advantage of.

IX. CONCLUSION

The aging experiments show that it can affect microservices and exacerbate their resource consumption. Also, Kubernetes probes do not show the ability to detect aging-related problems in the services monitored. This factor indicates that work in more effective approaches for early detection of microservices aging is needed. Besides, injecting faults to accelerate aging results in shorter periods required to evaluate the effectiveness of such mechanisms and allow timelier conclusions. The experimental campaign conducted with multiple faults allows observing the practical application of Kubernetes probes. Probes demonstrate significant shortcomings in detecting faults in the service probed, without any detection in the six faults injected.

Future work may explore more effective approaches to improving the probing mechanism or devising a methodology that can leverage other data sources, such as service data, to proactively identify aging or other kinds of faults present in the system.

REFERENCES

[1] Martin Fowler and James Lewis. (2014). *Microservices*. [Online]. Available: https://Martinfowler.com/articles/microservices.html
[2] P. Mell and T. Grance, "The NIST definition of cloud computing," Comput. Secur. Division, Inf. Technol. Lab., Nat., NIST, Gaithersburg, MD, USA, Tech. Rep. SP 800-145, 2011.
[3] Kubernetes. (2021). *What is Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[4] S. J. Vaughan-Nichols. (2020). *Kubernetes Jumps in Popularity*. [Online]. Available: https://www.zdnet.com/article/kubernetes-jumps-in-popularity/
[5] Kubernetes. (2020). *Configure Liveness, Readiness and Startup Probes*. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
[6] J. Flora, P. Goncalves, M. Teixeira, and N. Antunes, "My services got old! Can Kubernetes handle the aging of microservices?" in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2021, pp. 40–47.
[7] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Trans. Softw. Eng.*, vol. 47, no. 2, pp. 243–260, Feb. 2021.
[8] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proc. IEEE 26th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS*, Sep. 2018, pp. 223–236.
[9] Weaveworks. (2017). *Sock Shop : A Microservice Demo Application*. [Online]. Available: https://github.com/microservices-demo/microservices-demo
[10] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 1994, pp. 279–287.
[11] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *Proc. IEEE Int. Conf. Softw. Rel. Eng. Workshops (ISSRE Wksp)*, Nov. 2008, pp. 1–6.
[12] L. Li, K. Vaidyanathan, and K. S. Trivedi, "An approach for estimation of software aging in a web server," in *Proc. Int. Symp. Empirical Softw. Eng.*, 2002, pp. 91–100.
[13] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of software aging in a web server," *IEEE Trans. Rel.*, vol. 55, no. 3, pp. 411–420, Sep. 2006.
[14] J. Yue, X. Wu, and Y. Xue, "Microservice aging and rejuvenation," in *Proc. World Conf. Comput. Commun. Technol. (WCCCT)*, May 2020, pp. 1–5.
[15] F. Langner and A. Andrzejak, "Detection and root cause analysis of memory-related software aging defects by automated tests," in *Proc. IEEE 21st Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, Aug. 2013, pp. 365–369.
[16] J. Alonso, L. Belanche, and D. R. Avresky, "Predicting software anomalies using machine learning techniques," in *Proc. IEEE 10th Int. Symp. Netw. Comput. Appl.*, Aug. 2011, pp. 163–170.
[17] L. Vinícius, L. Rodrigues, M. Torquato, and F. A. Silva, "Docker platform aging: A systematic performance evaluation and prediction of resource consumption," *J. Supercomput.*, vol. 78, no. 10, pp. 12898–12928, Jul. 2022.
[18] B. Tola, Y. Jiang, and B. E. Helvik, "Model-driven availability assessment of the NFV-MANO with software rejuvenation," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 2460–2477, Sep. 2021.
[19] P. Kumari and P. Kaur, "A survey of fault tolerance in cloud computing," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 33, no. 10, pp. 1159–1176, 2021.
[20] M. Hasan and M. S. Goraya, "Fault tolerance in cloud computing environment: A systematic survey," *Comput. Ind.*, vol. 99, pp. 156–172, Aug. 2018.
[21] M. N. Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, "A survey of fault tolerance architecture in cloud computing," *J. Netw. Comput. Appl.*, vol. 61, pp. 81–92, Feb. 2016.
[22] A. Power and G. Kotonya, "A microservices architecture for reactive and proactive fault tolerance in IoT systems," in *Proc. IEEE 19th Int. Symp., World Wireless, Mobile Multimedia Networks (WoWMoM)*, Jun. 2018, pp. 588–599.
[23] A. Huff, M. Hiltunen, and E. P. Duarte, "RFT: Scalable and fault-tolerant microservices for the O-RAN control plane," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manag. (IM)*, May 2021, pp. 402–409.
[24] A. A. Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2019, pp. 1411–1415.

[25] G. M. Diouf, H. Elbiaze, and W. Jaafar, "On Byzantine fault tolerance in multi-master Kubernetes clusters," *Future Gener. Comput. Syst.*, vol. 109, pp. 407–419, Aug. 2020.
[26] A. Javed, K. Heljanko, A. Buda, and K. Framling, "CEFIoT: A fault-tolerant IoT architecture for edge and cloud," in *Proc. IEEE 4th World Forum Internet Things (WF-IoT)*, Feb. 2018, pp. 813–818.
[27] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, and M. Kihl, "Impact of ETCD deployment on Kubernetes, istio, and application performance," *Softw., Pract. Exper.*, vol. 50, no. 10, pp. 1986–2007, Oct. 2020.
[28] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017, pp. 195–216.
[29] J. Flora and N. Antunes, "Studying the applicability of intrusion detection to multi-tenant container environments," in *Proc. 15th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2019, pp. 133–136.
[30] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah, "Securing microservices," *IT Prof.*, vol. 21, no. 1, pp. 42–49, Jan. 2019.
[31] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35–41, May/Jun. 2016.
[32] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52976–52996, 2019.
[33] Kubernetes. (2020). *Resource Management for Pods and Containers*. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
[34] K. Jackson. (2020). *Types of Kubernetes Events*. [Online]. Available: https://www.bluematador.com/blog/kubernetes-events-explained
[35] B. M. Timofte. (2018). *Kubernetes Readiness & Liveliness Probes-Best Practices*. [Online]. Available: https://medium.com/metrosystemsro/kubernetes-readiness-liveliness-probes-best-practices-86c3cd9f0b4a
[36] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Sec. Comput.*, vol. 1, no. 1, pp. 11–33, Jan./Mar. 2004.
[37] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proc. 16th IEEE Symp. Reliable Distrib. Syst.*, Oct. 1997, pp. 72–79.

JOSÉ FLORA (Graduate Student Member, IEEE) received the M.Sc. degree in information security from the University of Coimbra, Portugal, in 2019, where he is currently pursuing the Ph.D. degree in informatics engineering. His research interests include information and software security, particularly software containers and microservices security, intrusion detection and intrusion tolerance, and security services for cloud computing.

PAULO GONÇALVES received the bachelor's degree from the University of Coimbra, Portugal, in 2020, where he is currently pursuing the M.Sc. degree in informatics engineering. His research interests include intrusion detection and tolerance, reverse engineering, and genetic algorithms.

MIGUEL TEIXEIRA received the bachelor's degree from the University of Coimbra, in 2020, where he is currently pursuing the M.Sc. degree in informatics engineering with specialization in software engineering. His research interests include development and automation of microservices related systems.

NUNO ANTUNES (Member, IEEE) received the Ph.D. degree in information science and technology from the University of Coimbra, in 2014. He has been with the Centre for Informatics and Systems of the University of Coimbra (CISUC), since 2008. He is an Assistant Professor with the University of Coimbra. His research interests include testing, fault injection, vulnerability injection and benchmarking, which are applied to the assessment of the dependability and security of intelligent systems, virtualized environments, intrusion detection systems, web services, web and mobile applications, and data management systems. He is a member of the IEEE Computer Society.