软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(5):1231−1255 [doi: 10.13328/j.cnki.jos.006231]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于混沌工程的微服务韧性风险识别和分析*

殷康璘, 杜庆峰

(同济大学 软件学院,上海 201804)

通讯作者: 杜庆峰, E-mail: du_cloud@tongji.edu.cn

摘 要: 微服务架构近年来已成为互联网应用所采用的主流架构模式.然而与传统的软件架构相比,微服务架构更加复杂的部署结构使其面临更多能够导致系统发生故障的潜在威胁,且微服务架构系统故障的症状也更加多样化.在可靠性等一些传统的软件度量已不能充分体现微服务架构系统故障应对能力的情况下,微服务的开发者们开始使用“韧性(resilience)”一词描述微服务架构系统的故障应对能力.为了提高微服务架构系统的韧性,开发者往往需要针对特定的系统环境扰动因素设计应对机制.如何判断一个系统环境扰动因素是否为影响微服务系统韧性的风险因素,以及如何在系统运行发布之前尽可能多地寻找到这些潜在的韧性风险,都是微服务架构系统开发过程中待研究的问题.在先前研究中提出的微服务韧性度量模型的基础上,结合混沌工程,提出了针对微服务架构系统的韧性风险识别和分析方法.韧性风险的识别方法通过不断地向微服务架构系统引入随机系统环境扰动并观察系统服务性能的变化,寻找系统潜在的韧性风险,大幅度减少了软件风险识别过程中的人力成本.对于识别到的韧性风险,通过收集执行混沌工程过程中的系统性能监控数据,韧性风险分析方法将利用因果搜索算法构建出各项系统性能指标之间的影响链路,并将可能性较高的链路提供给运维人员,作为进一步分析的参考.最后,通过在一个微服务架构系统上实施的案例,研究展示了所提出的韧性风险识别和分析方法的有效性.

关键词: 微服务;韧性;软件风险识别;混沌工程

中图法分类号: TP311

中文引用格式: 殷康璘,杜庆峰.基于混沌工程的微服务韧性风险识别和分析.软件学报,2021,32(5):1231−1255. http://www.jos.org.cn/1000-9825/6231.htm

英文引用格式: Yin KL, Du QF. Microservice resilience risk identification and analysis based on chaos engineering. Ruan Jian Xue Bao/Journal of Software, 2021,32(5):1231−1255 (in Chinese). http://www.jos.org.cn/1000-9825/6231.htm

Microservice Resilience Risk Identification and Analysis Based on Chaos Engineering

YIN Kang-Lin, DU Qing-Feng

(School of Software Engineering, Tongji University, Shanghai 201804, China)

Abstract: Microservice architecture has already become the mainstream architecture pattern of Internet applications in recent years. However, compared with traditional software architectures, microservice architecture has a more sophisticated deployment structure, which makes it have to face more potential threats that make the system in fault, as well as the greater diversity of fault symptoms. Since traditional measurements like reliability cannot fully show a microservice architecture system’s capability to cope with failures, microservice developers started to use the word “resilience” to describe such capability. In order to improve a microservice architecture system’s resilience, developers usually need to design specific mechanisms for different system environment disruptions. How to judge whether a system environment disruption is a risk to microservice resilience, and how to find these resilience risks as much as possible

* 基金项目: 国家自然科学基金(U1934212); 国家重点研发计划(2020YFB2103300)

Foundation item: National Natural Science Foundation of China (U1934212); National Key Research and Development Program of China (2020YFB2103300)

本文由“面向持续软件工程的微服务架构技术”专题特约编辑张贺教授、王忠杰教授、陈连平研究员和彭鑫教授推荐.

收稿时间: 2020-07-10; 修改时间: 2020-10-26, 2020-12-15; 采用时间: 2021-01-18; jos 在线出版时间: 2021-02-07

before the system is released, are the research questions in microservice development. According to the microservice resilience measurement model which is proposed in authors' previous research, by integrating the chaos engineering practice, resilience risk identification and analysis approaches for microservice architecture systems are proposed. The identification approach continuously generates random system environment disruptions to the target system and monitors variations in system service performance, to find potential resilience risks, which greatly reduces human effort in risk identification. For identified resilience risks, by collecting performance monitoring data during chaos engineering, the analysis approach uses the causality search algorithm to build influence chains among system performance indicators, and provide chains with high possibility to system operators for further analysis. Finally, the effectiveness of the proposed approach is proved by a case study on a microservice architecture system.

Key words: microservice; resilience; software risk identification; chaos engineering

微服务架构(microservice architecture)是由马丁·福勒所提出的一种新的软件架构模式[1],它将一个单体的软件系统拆分为若干可独立运行、部署的微服务(microservices).微服务架构将软件功能变更的规模控制在一个微服务内部,且不影响其他的微服务,大幅减少了软件功能迭代过程中系统重新构建、测试、部署的成本,因此,采用微服务架构成为了 DevOps 开发模式中常用的手段[2].近年来,微服务架构已成为许多互联网公司会使用的一种主流软件架构模式[3−5].

相对于早年面向服务架构的软件系统,采用微服务架构的软件系统(以下简称微服务架构系统)对服务的划分更细粒度化,并且通常会使用容器技术提高系统资源的利用率,以致微服务架构系统部署结构更为复杂.受此影响,微服务架构系统会面临更多非软件设计缺陷因素(如服务器意外宕机、网络不稳定等)所引发的软件系统故障[6].除了软件系统故障以外,软件系统的升级、系统部署配置(如微服务的冗余备份配置、虚拟机的资源分配)的动态变更、未预期的工作负载等情况,均可能导致微服务架构系统不能正常提供其服务[7].

在传统的软件系统质量度量中,描述系统应对故障能力的度量指标有可用性、可靠性、容错性等[8],这些度量指标往往将系统的状态分为"可用/可靠"和"不可用/不可靠"两种.但是,近年来对云计算故障模式的相关研究表明[7,9]:故障除了能直接导致系统服务本身的不可用之外,也可能会使系统服务质量(性能)受到严重影响,但是系统服务仍处于可访问状态.在这种情况下,可用性、可靠性这一类指标并不能完全体现出一个微服务架构系统在故障发生时其系统服务性能受到影响的严重程度.例如,一个系统在情况 *A* 下系统服务的平均响应时间从 3s 延长至 5s,在情况 *B* 下系统服务的平均响应时间从 3s 延长至 12s,假设情况 *A* 和情况 *B* 持续的时长相同,那么系统在情况 *A* 和情况 *B* 下的可靠性也是相同的.但是很明显,系统在情况 *B* 下服务性能受影响的程度比在情况 *A* 下严重.另一方面,现有对软件系统服务性能的评估方法主要以性能测试为主,通过性能测试可以识别出系统在一定服务压力下体现出来性能设计缺陷.但是系统在故障发生时,系统服务性能受到的影响并不会在性能测试中验证.

基于上述原因,微服务架构系统的相关研究人员开始使用"韧性(resilience)"一词表示系统处理故障的能力[10,11]("resilience"一词在不同学术领域中有多种翻译,通常被翻译为"弹性""韧性",而国内计算机领域目前对 resilience 的翻译尚未确定.由于"弹性"一词早已在云计算中被用来形容软件系统的伸缩性和可扩展性,本文使用"韧性"一词作为 resilience 的翻译).为了提高微服务架构系统的韧性,负载均衡、熔断机制、心跳检测等一些常用的系统容错机制[12]被开发人员和架构设计人员应用在系统上.

在计算机领域中,目前还没有对软件韧性有统一的定义.根据其他领域研究中对韧性的定义[13]以及韧性这一概念在微服务架构系统开发者中被使用的情况,本文作者在先前的研究工作[14]中从服务性能的角度将微服务架构系统的韧性定义为:"一个微服务架构系统在系统环境扰动发生并导致其服务性能下降后,维持其服务性能在一个可接受的水准,并快速将服务性能恢复至正常状态的能力".其中,系统环境扰动(disruption)是其他领域韧性研究中的一个通用概念,意为影响系统(非特指软件系统)功能正常运作的事件.在微服务架构系统中,系统环境扰动既包括软件系统的内部组件故障,也包括上文中所提及的诸如系统升级、配置变更等使微服务架构系统产生"变更"的事件.在上述定义的基础上,该研究提出了微服务韧性度量模型 MRMM(microservice resilience measurement model),该模型将微服务架构中有关韧性的概念进行概念建模,并给出用于度量系统环境扰动发生时服务性能变化的 3 个度量维度,以评估其对微服务架构系统服务性能的影响程度.

通过设立由 MRMM 的 3 项度量指标构成的服务韧性目标,可以描述出一个微服务架构系统预期达到的服务韧性;随后,微服务架构系统的开发人员将在各种可能的系统环境扰动中寻找出会超出韧性目标阈值的扰动,将其认定为威胁微服务韧性的软件风险(以下简称为韧性风险),并为其设计系统应对机制[15].在传统的软件风险分析过程中,对软件风险的识别通常采用头脑风暴、专家经验等人为分析方法,然而在微服务架构系统中,随着微服务数量的增加以及服务之间调用关系的复杂化,根据微服务架构系统中各类型系统资源可能发生的环境扰动事件类型,人为地列举出所有可能发生的具体系统环境扰动(如某一个服务存在一种扰动类型,就需要穷尽目标系统的各个服务在发生这种扰动后可能的情况)并逐个验证这些扰动是否会产生严重的服务降级,显然会消耗大量的人力成本以及时间成本.对于识别到的韧性风险,现有的故障诊断和分析方法需要对目标系统的所有通信过程植入监控代码,或对历史性能数据人工地标注系统正常异常与否,将花费大量额外的系统开发成本或人工成本.此外,没有统一的韧性度量方法,使得微服务架构系统的研究人员难以界定服务性能受系统环境扰动影响的严重程度,并在迭代过程中选择需要优先处理的韧性风险.综上所述,微服务架构系统的韧性风险识别过程中存在着以下两个问题.问题 1:如何使用较少的人力和时间成本识别出目标微服务架构系统的韧性风险?问题 2:如何分析识别到的韧性风险对目标微服务架构系统的影响?

针对上述问题,本文提出了微服务韧性风险的识别和分析方法,其整体流程如图 1 所示.首先,本方法根据 MRMM 模型中的韧性度量指标为目标微服务系统中的服务设立韧性目标;随后,基于混沌工程的方法执行若干次混沌实验,在每次混沌实验中,以随机的方式生成系统环境扰动,通过比较实验中系统环境扰动产生的服务降级是否超出服务韧性目标的阈值范围,识别出目标系统中的韧性风险.针对每一个被识别的韧性风险,为了免去对微服务架构系统的额外开发成本和对性能数据的人工标注成本,本文通过因果关系搜索算法无监督地分析实验结果数据中各系统性能指标之间的因果关系,并给出可能的韧性风险影响链路.

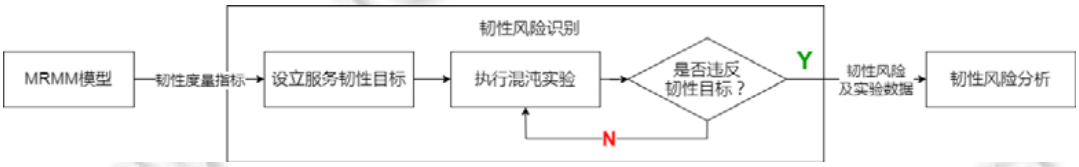


Fig.1 Process of microservice resilience risk identification and analysis

图 1 微服务韧性风险的识别和分析过程

本文第 1 节概述本文的相关研究.第 2 节介绍基于混沌工程的韧性风险识别方法.第 3 节介绍针对微服务架构系统的韧性风险分析方法.第 4 节以一个微服务架构系统 Sock-Shop 为例,对本文提出的方法进行案例研究.第 5 节进一步分析案例研究的实验结果,总结本文的工作,并提出下一步的研究计划.

1 相关工作

在学术研究中,韧性最早由生态学家 Holling 于 1973 年所提出,用以表示一个生态系统响应外部干扰并恢复由外部干扰所带来的损害的能力[16].随后,韧性这一概念逐渐被引入至社会-生态系统、经济学、组织管理、城市规划等多个领域[17,18].在软件/计算机领域中,韧性的概念虽然在微服务架构诞生之前就已被提出,但计算机领域有关韧性的研究相对于其他领域仍处于初步阶段.Laprie 于 1992 年在可靠性(dependability)相关定义的综述中提出了韧性的概念,并将韧性定义为软件系统对故障的恢复能力[19];随后,于 2008 年提出将韧性作为可靠性(reliability)的引申研究,重新将韧性定义为:在软件系统在面对系统环境、配置等因素变化时,系统维持服务发布状态的能力[20].2012 年,由科因布拉大学和纽卡斯尔大学建立的 AMBER(assessing, measuring, and benchmarking resilience)组织回顾了各文献对韧性的定义,并阐述了韧性在云计算中的必要性以及与软件韧性的相关研究[21].一些针对软件/计算机韧性的研究[22−24]基本都通过应用其他领域的定义来说明韧性这一概念,并尝试解释韧性与可靠性、容错性等软件质量特性的关系.

在微服务架构被提出的几年内,微服务架构韧性的重要性就已在一些应用书籍[10,25]以及学术研究[11]中被

提出.与软件/计算机整个领域的现有韧性研究相同,目前绝大部分提到微服务架构韧性的研究仅仅提到了韧性的概念,并基本以现有韧性机制(如负载均衡、熔断机制等)的优化研究为主,包括微服务的健康管理[26]、负载均衡[27]、故障恢复机制[28]、故障发现等[29].也有少数针对微服务韧性展开的相关理论研究.Heorhiadi 等人提出了一种微服务架构的韧性测试框架[30];Thomas 和 Andre 提出了用于微服务架构性能和韧性基线分析的元模型[31],Giedrimas 根据现有的韧性机制总结了微服务架构韧性设计中需要考虑的几个方面[32],Andre 和 Aleti 提出了一种结合了工作负载和故障注入的微服务架构韧性基线验证框架[33],Michal 和 Marian 比较了单体应用和微服务架构之间在性能和韧性之间的差异[34].然而,现有微服务韧性的研究中并未有从软件风险角度讨论韧性的研究.

软件风险评估(software risk evaluation)是软件工程中的重要过程之一,其中,软件风险识别是软件风险评估过程中的首要步骤[35].主流的软件风险评估方法有故障树分析法(fault tree analysis)[36]、错误用例法(misuse case)[37]、威胁建模(threat modeling)[38]以及 FMEA 法[39].上述的评估方法在软件风险识别阶段均使用人为方法识别软件风险,通过参考专家经验以及类似的历史项目或者头脑风暴来列出可能威胁软件本身以及软件开发过程中潜在的风险.其中,威胁建模在风险识别阶段引入了 STRIDE 模型[38],通过枚举不同的风险类型启发风险分析人员找到软件项目的潜在风险.近年来,对软件风险导出方法的研究也主要以针对不同领域软件的优化为主,软件风险的识别仍然使用人为方法[40,41].

混沌工程[42]是由 Netflix 公司于 2017 年提出的一种用于验证微服务系统对故障应对能力的实验方法.通过随机地对目标系统进行故障注入,观察系统服务是否被故障影响.Netflix 针对自身的应用的关键服务——视频播放服务,以每秒视频的启动次数(video-stream start per second,简称 SPS)作为度量指标,通过观察故障注入后视频播放服务的 *SPS* 值是否会低于指定阈值,判断注入的故障是否为潜在系统风险.目前,在计算机领域有关混沌工程的学术研究极少,在 SCI、EI、DBLP 等数据库中对“Chaos Engineering”一词进行检索,仅能搜索到少量与 Netflix 提出的混沌工程概念一致的相关文献.在提出混沌工程的概念后,Netflix 又针对混沌工程的商业价值[43]、测试平台[44]及混沌实验的执行过程[45]进行了进一步阐述.Zhang 等人针对 Java 虚拟机提出了 ChaosMachine 框架[46],该框架能够在混沌实验的过程中.通过添加 Java 注释的方式实现在 Java 代码中插入异常(exception)抛出行为,以验证 Java 服务捕获并处理异常的能力.Jesper 等人针对容器环境提出了 ChaosOrca 框架[47],实现了对部署微服务环境的容器的系统调用进行故障注入.Kennedy 等人利用混沌工程的方法随机地对云服务进行安全性攻击,以发现云服务中的数据安全问题.目前,混沌工程已被业界认可为一种可实践的方法论[48−50],并已有一些可用于实践的工具,如 Netflix 的 ChaosMonkey、阿里巴巴的 Chaosblade、混沌工程服务 Gremlin 以及开源项目 Chaos-toolkit 等.这些工具目前的主要功能为实现特定的系统环境扰动,并提供对应混沌实验文件模板或指令给测试人员,而有关混沌实验的具体设计过程并没有相关实现.本文的案例研究中,也将利用上述工具实现的系统环境扰动发现目标系统潜在的韧性风险.

故障诊断是用于分析软件风险对软件系统影响的一种重要方法.现有针对分布式服务架构的软件系统的故障诊断方法可以分为基于监督学习的方法和基于系统调用关系的方法.

- 基于监督学习的方法[51−54]需要在一个已标签过的数据集上进行机器学习模型的训练,训练后的模型能够判断当前系统中是否存在某一种类型的异常.目前,基于监督学习的故障诊断方法需要大量收集系统的历史数据,并且通常仅能判断目标系统是否存在某一特定类型的故障.
- 基于系统调用关系的方法将首先构建出系统服务和系统资源之间的依赖关系图,并根据依赖关系定位系统故障的根因.其中,系统服务和资源的依赖关系通常参考已有的经验知识[55−58],或者通过在代码中插桩获取服务或组件之间调用关系[59−62].这一过程需要人工识别服务和资源之间的依赖关系或者在现有系统上开发特定的工具捕获依赖关系.

为了能够仅通过系统性能数据构建出微服务架构系统的依赖关系图,陈鹏飞等人先后提出了 CloudRanger[63]、Microscope[64]、CauseInfer[65]这 3 种根因诊断模型.上述 3 种模型均使用了因果搜索算法分析目标微服务系统的系统资源和服务之间的因果关系并构建因果关系图,且通过实验证明了因果搜索算法在对微服务系统的故障诊断上相对于现有故障诊断方法在准确率和效率上具有显著的优势,且无需对系统性能数

据标注标签.参考上述研究工作,本文在韧性风险的分析过程中同样使用了因果关系搜索算法来构建目标微服务系统中性能指标之间的依赖关系.

本文作者在先前的研究工作[14]中提出了微服务韧性度量模型 MRMM,该模型使用了性能降级程度、降级恢复时间和降级损失这 3 个度量维度度量了一次系统环境扰动产生的服务降级中系统性能的变化,以体现系统环境扰动对微服务架构系统的影响程度,如图 2 所示.以 MRMM 为基础,该研究提出了一种韧性需求的表示方法,通过目标导向的需求模型体现出微服务架构中韧性目标、韧性风险和韧性机制之间的关系.然而,如何识别出一个微服务架构系统中的韧性风险并对其进行分析,并没有在该研究的讨论范围内.

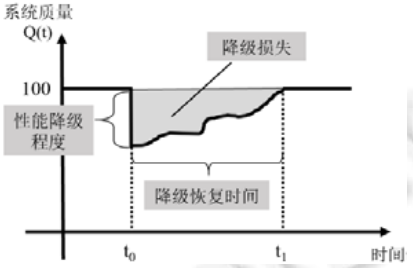


Fig.2 Three resilience metrics in MRMM

图 2 MRMM 中的 3 个韧性度量指标

2 微服务架构系统韧性风险的识别方法

基于混沌工程的实践方法和微服务韧性度量模型 MRMM[14],本文提出了针对微服务架构系统的韧性风险识别方法.相对于传统的软件风险评估模型中人为识别软件风险的方法,该方法将利用混沌工程工具中已实现的系统扰动事件类型,以随机的方式自动构建出大量混沌实验,持续地对目标微服务架构系统进行混沌实验,并以预先设立的韧性度量目标判断某种系统环境扰动是否为韧性风险,实现了一种针对韧性风险的发现策略,并免去了人为穷尽并验证各服务中不同系统资源类型的各种环境扰动对目标系统影响的人工成本.具体的混沌实验实例详见本文第 4.2 节的案例研究部分.

2.1 假设与前提

本文所提出的微服务系统韧性风险识别方法具有以下假设和前提.

(1) 目标微服务架构系统遵循 DevOps 的开发模式,系统拥有完整的自动化构建、部署体系.

(2) 目标微服务架构系统已完成了首次迭代,并有一个可部署执行的版本.

(3) 目标微服务架构系统中,所有系统服务的实时服务性能数据可以通过工具采集.

(4) 目标微服务架构系统由于本身的性能设计缺陷导致的服务性能下降不在本文的研究范围内.

(5) 本文将仅考虑单个系统环境扰动对微服务架构系统的影响,对于多种扰动对服务的组合影响,将在本文的后续研究中进一步探讨.

2.2 韧性风险识别方法

本文提出的韧性风险识别方法将包含以下几个步骤.

(1) 确立服务性能基线

通过设立由 MRMM 的 3 项度量指标构成的服务韧性目标,可以描述出一个微服务架构系统预期达到的服务韧性.当一个服务发生的某次服务降级所测量得到的服务韧性超出了其服务韧性目标的阈值范围,导致该次服务降级的扰动将被识别为微服务架构的韧性风险.MRMM 中的各项韧性度量指标均通过计算服务实际性能与服务性能基线的差异得到,因此在韧性风险识别过程的开始阶段,需要对目标微服务架构系统所提供的每个系统服务建立服务性能基线,以判断服务是否处于服务降级状态.确立服务性能基线的过程中包含两个步骤.

a) 确定服务性能指标.

根据服务类型的不同,服务性能的关注重点也会不同.因此,每一个系统服务都需要确定其关键服务性能指标.服务性能指标的选择可以参考现有 IT 系统性能标准(如 SPEC[66]、TPC[67,68]、ETSI GS[69])、性能评估数据集[70,71]以及其他服务度量指标选择相关研究[72]中常见的性能指标.考虑到微服务架构系统的最终目标是为互联网公司获取盈利,一些特定的服务需要设立与商业需求有关的性能指标.例如,Netflix 将每秒用户视频的播放数量作为一种需要检测的关键性能指标[43].

b) 根据服务性能指标确立服务性能基线.

在确定服务性能指标之后,对每一个服务的服务性能指标建立对应的服务性能基线.根据性能指标类型的不同,性能基线既可能是一个定值(如服务响应时间的基线值通常为一个定值),也可能随事件变化动态取值(如销售服务的每秒成功交易量的基线值会随着销售淡旺季变化而变动).定值的性能基线的设立方式有统计历史运行数据(如服务处于正常状态时的性能数值)、领域专家给出经验值、参考相关标准(如上文提到的 IT 系统性能标准)给出的建议值等,动态的性能基线则可以在历史服务性能数据上使用时间序列数据处理算法(如 EWMA、ARMIA、LSTM 等)计算服务在某个时段中若正常运行其性能的预估值来得到.

(2) 设立服务韧性目标

服务韧性目标是用于判断系统环境扰动是否为韧性风险的基准,其体现了系统环境扰动发生后系统服务的服务降级程度的可接受范围.服务韧性目标将按照 MRMM 中的 3 个韧性度量指标进行设立.各个系统服务的服务韧性目标值的设立将依据各个服务的实际业务需求,其中没有意义的度量维度则不需要设立对应的目标值(例如,以系统响应时间作为性能指标的服务不需要设立降级损失的服务韧性目标).

(3) 设置混沌实验参数

微服务架构韧性风险的识别过程中,需要对混沌实验进行参数限制以控制整个识别过程所花费的时间.混沌实验的参数包含:

① 实验次数:一次韧性风险识别过程中将执行多少次混沌实验.

② 实验最大时长:每次混沌实验最多持续多少时间,若一次混沌实验的实验时间超过设置的最大时长,该次混沌实验将被强制终止.

(4) 进行混沌实验

混沌实验将通过随机地生成系统环境扰动,并将其引入至正在运行的目标微服务架构系统中.在执行混沌实验的过程中,为了保证每一次混沌实验不会被前一次混沌实验中引入的系统环境扰动所影响,混沌实验将利用自动化构建工具将目标微服务系统进行重新构建、部署,并在实验结束后销毁实验环境.每一次混沌实验中,将包括以下两个随机因素.

- 系统压力场景:在混沌实验中,目标微服务架构系统将承受一定的业务压力,从而使由于系统环境扰动产生的服务降级更加明显.而为了使实验环境能够更加贴近系统真实运行环境,系统业务压力的模拟将参考系统在真实运行环境中可能面临的业务压力场景.每一次混沌实验,将随机地选取一个压力场景进行业务压力模拟.
- 系统环境扰动:混沌实验中,系统环境的扰动将按照以下方式生成:首先,系统环境扰动的生成工具将随机地选择发生系统环境扰动的资源类型;随后,系统环境扰动的生成工具将在实验环境中指定资源类型的实例中随机选取一个作为系统环境扰动发生的具体位置;最后,在该资源类型中可能发生的系统环境扰动事件中随机地选取一个事件作为在混沌实验中发生的系统环境扰动事件.

在执行混沌实验的过程中,除了系统压力场景的模拟以及系统环境的扰动之外,还需要对目标系统实时地监控并进行数据采集.数据采集的对象包括两部分:① 系统服务的服务性能数据,用以实验结束后度量实验中引入系统环境扰动后产生的服务降级;② 系统资源性能数据(如 CPU 使用率、网络流量等).

(5) 韧性风险识别

在所有的混沌实验结束后,混沌实验中收集到的服务性能数据将与混沌实验前预设的服务性能基线数据进行比较,判断系统环境扰动过程中系统服务是否发生了服务降级.若系统服务发生了服务降级,则根据

MRMM 中的 3 个韧性度量维度对实验中发现的服务降级进行度量,并与预设的服务韧性目标比较:如果服务降级的程度超出了服务韧性目标所预设的阈值,则说明这次混沌实验中所引入的系统环境扰动是严重影响系统服务性能的韧性风险,该次混沌实验中生成收集得到系统服务性能数据和系统资源性能数据将作为韧性风险分析的依据.

3 微服务架构系统的韧性风险分析

如本文引言中所述,本文提出的韧性风险分析方法将避免对微服务架构系统本身的额外开发成本,以无监督的方法分析一个韧性风险如何影响目标系统并引发服务降级.该方法将以被识别的韧性风险所对应的混沌实验中收集的系统性能数据为依据,通过因果搜索算法构建系统性能指标之间的因果关系图,随后根据系统性能数据的上升和下降变化对因果关系图中的各条边赋予权重,最终输出若干由系统性能指标构成的韧性风险的影响链路,这些影响链路将按照因果关系边的权重排序.微服务架构系统的运维人员可以根据韧性风险的影响链路中涉及到的系统性能指标设计对应的系统优化方案.韧性风险分析方法的整体流程如图 3 所示.

Fig.3 Process of resilience risk analysis

图 3 韧性风险分析过程

3.1 混沌实验数据收集

对于每一个被识别的韧性风险,其对应的混沌实验执行过程中的服务性能和资源性能的历史数据将作为韧性风险如何产生服务降级的分析依据.本文提出的韧性风险分析方法所分析的目标为系统各项性能指标之间的因果变化关系,因此会在每一次混沌实验中将通过监控工具收集目标系统在该次混沌实验中的系统性能数据集{*timestamp*,K_1,K_2,…,K_n},其中,*timestamp* 为时间戳,K_1,…,K_n 为在该时间戳下的各项系统性能指标.图 4 为一个样例的混沌实验数据集,其中,第 1 列为每一行性能数据对应的 Unix 时间戳,第 2 列开始的每一列表示某一项系统性能指标(如第 3 列为 cart 服务的平均请求延时,单位为 s),某一行某一列的数据表示该行第 1 列时间戳下捕获到的该列所对应的系统性能指标的实际值.

timestamp	service/carts/qps(2xx)	service/carts/latency	service/catalogue/qps(2xx)	service/catalogue/latency	service/front-end/qps(2xx)	service/front-end/latency	service/orders/qps(2xx)	service/orders/latency	service/payment/qps(2xx)	service/payment/latency	service/shipping/qps(2xx)	service/shipping/latency	service/user/qps(2xx)	service/user/latency
1561014180	128.1667	0.291587	19.76667	0.003963	18.92722	0.355426	1.566667	0.034354	2.166667	0.000458	1.966667	0.001335	129.0452	0.039459
1561014190	128.1667	0.312908	19.76667	0.003963	52.10453	0.445334	3.9	0.033161	2.166667	0.000458	1.966667	0.001335	129.0452	0.039915
1561014200	128.1667	0.312908	36.16667	0.004194	100.1044	0.193352	6.6	0.034444	2.166667	0.000458	1.966667	0.001335	209.5693	0.008005
1561014210	90.6	0.024261	36.16667	0.004194	151.9181	0.159688	13	0.050783	18.56667	0.000557	17.5	0.00134	209.5683	0.008005
1561014220	90.6	0.024261	36.16667	0.004194	153.9599	0.062557	20.19969	0.055147	18.56667	0.000557	17.5	0.00134	209.5683	0.008005
1561014230	90.6	0.024261	35.16667	0.005079	161.7592	0.086918	22.66635	0.056264	18.56667	0.000557	17.5	0.00134	238.8	0.016664
1561014240	96.03333	0.017041	35.16667	0.005079	166.2194	0.076337	25.29969	0.049644	26.46667	0.000851	26.03333	0.001357	238.8	0.016664
1561014250	96.03333	0.017041	35.16667	0.005079	169.132	0.076965	26.96705	0.045113	26.46667	0.000851	26.03333	0.001357	238.8	0.016664
1561014260	96.03333	0.017041	35.46667	0.005835	172.6653	0.060672	28.13372	0.044613	26.46667	0.000851	26.03333	0.001357	266.1578	0.011583
1561014270	103.9	0.022341	35.46667	0.005835	173.868	0.062241	31.53372	0.051343	32.9	0.000762	33.1	0.001446	266.1578	0.011583
1561014280	103.9	0.022341	35.46667	0.005835	173.7014	0.065336	34.03333	0.054254	32.9	0.000762	33.1	0.001446	266.1578	0.011583
1561014290	103.9	0.022341	34.7	0.006285	174.0014	0.074948	34.26667	0.056331	32.9	0.000762	33.1	0.001446	275.8759	0.013897
1561014300	101.7333	0.019755	34.7	0.006285	174.6333	0.080162	33.83333	0.066541	33.76667	0.001599	33.46667	0.001364	275.8759	0.013897
1561014310	101.7333	0.019755	34.7	0.006285	173.5	0.140679	33.46643	0.102864	33.76667	0.001599	33.46667	0.001364	275.8759	0.013897
1561014320	101.7333	0.019755	34.79884	0.024697	170.7	0.224427	33.06643	0.121268	33.76667	0.001599	33.46667	0.001364	265.2	0.056465
1561014330	101.5333	0.028003	34.79884	0.024697	170.1333	0.245268	31.86643	0.130628	32.93333	0.00159	33.2	0.001096	265.2	0.056465
1561014340	101.5333	0.028003	34.79884	0.024697	171.6	0.186644	32.30023	0.097716	32.93333	0.00159	33.2	0.001096	265.2	0.056465
1561014350	101.5333	0.028003	34.76783	0.004748	173.5579	0.105442	32.6669	0.079354	32.93333	0.00159	33.2	0.001096	267.0489	0.020323
1561014360	104.6667	0.01905	34.76783	0.004748	173.6	0.10332	35.03357	0.061927	34.56667	0.000656	33.3	0.001107	267.0489	0.020323
1561014370	104.6667	0.01905	34.76783	0.004748	174.3333	0.092865	34.29978	0.054383	34.56667	0.000656	33.3	0.001107	267.0489	0.020323

Fig.4 A sample chaos experiment dataset

图 4 混沌实验数据集样例

3.2 因果关系图构建

基于在混沌实验中收集的系统性能数据集,本文提出的韧性风险分析方法将首先通过因果搜索算法构建性能指标之间的因果关系图.因果关系图为由一系列表示性能指标的节点和表示性能指标之间因果关系的边组成,因果关系图中所有的边仅表示性能指标之间的直接因果关系,不包含间接因果关系.若某个性能指标 X 影响了另一个性能指标 Y,则用有向边 $X\to Y$ 表示.如果算法能明确两个性能指标 X,Y 之间存在因果关系但不能明确是 X 影响了 Y 还是 Y 影响了 X,则用无向边 $X—Y$ 表示.

本文中使用的因果搜索算法参考了 Sprite 和 Glymour 所提出的 PC 算法[73]的主要思想,最终,算法将输出一张包含有向边和无向边的因果关系图.因果关系构建算法的具体流程如算法 1 所示,其主要包含两个步骤:因果关系确立以及因果方向确立.在因果关系确立阶段,算法首先构建以数据集内性能指标为节点的完全图;随后,对于任意两个性能指标 X,Y,算法通过判断 X 和 Y 是否条件独立来确定 X 和 Y 之间是否存在因果关系.若 X 和 Y 条件独立,说明 X 和 Y 之间不存在直接的因果关系,则在完全图中删除对应的边;最后得到初步的仅包含无向边的因果关系图.在因果方向确立阶段,算法将先根据 d-分隔原则确定因果关系图中所有的 $X\to Y\leftarrow Z$ 的结构,即 V-Structure;随后,根据在 V-Structure 中已经确定的有向边和一些逻辑推断规则(算法 1 伪代码的第 15 行~第 21 行),算法将进一步确定现有的因果关系图中部分剩余无向边的方向.例如,在已经确定图中所有 $X\to Y\leftarrow Z$ 结构的情况下,若发现结构 $A—B\leftarrow C$,则可以直接判断边 $A—B$ 的方向为 $A\leftarrow B$.

算法 1. 因果关系图构建算法.

输入:混沌实验数据集 D,数据集 D 中的监控指标集合 V.

输出:因果关系图 G.

1. //因果关系确立
2. 以 V 中的元素作为节点,构建完全图 G
3. $n\leftarrow V$ 中元素的数量
4. **for** k from 0 to n:
5. **for each** X,Y 为 G 中的两个相邻节点,且邻接于 X 的节点数量大于 k:
6. **for each** 集合 $W\subset V\backslash\{Y\}$ 为邻接于 X 的节点集合,且 W 的元素数量为 k:
7. **if** X,Y 条件独立于 W **then**:
8. 去除 X 与 Y 之间的边
9. 记录 W
10.
11. //因果方向确立
12. **for each** G 中相邻的 3 个节点 $X—Y—Z$:
13. **if** Y 不存在于所有使得 X 于 Z 独立的节点集合中:
14. 把 G 中 $X—Y—Z$ 的方向修改为 $X\to Y\leftarrow Z$
15. **for each** G 中相邻的 3 个节点 X,Y,Z:
16. **if** $X\to Y—Z$:
17. 把 $Y—Z$ 的方向修改为 $Y\to Z$
18. **if** $X—Y$ and $X\to Z\to Y$:
19. 把 $X—Y$ 的方向修改为 $X\to Y$
20. **if** $X—Y$ and 存在另一变量 L,使得 $X—Z\to Y$ 且 $X—L\to Y$:
21. 把 $X—Y$ 的方向修改为 $X\to Y$
22. **return** G;

在确定两个性能指标 X,Y 之间是否存在因果关系的过程中,基于 d-分隔原则,需要判断 X 和 Y 在其他性能指标为给定值的情况下是否条件独立.由于微服务架构系统中通过监控工具获取的系统性能数据大部分为连

续值,为了判断连续数据的条件独立性[74],本文使用了如下方法.

建立零假设 H_0:性能指标 X,Y 在给定条件 C 下的偏相关系数$\rho_{XY,C}$=0.其对应的备择假设 H_1 为:X,Y 在给定条件 C 下的偏相关系数$\rho_{XY,C}$≠0.通过 F 检验验证 H_0 是否成立:若 H_0 成立,则说明监控指标 X,Y 在给定条件 C 下条件独立,其实际意义为 X,Y 之间没有因果关系或者 X,Y 经由 C 产生了间接的因果关系.零假设 H_0 在 F 检验下的显著性水平 $z(\rho_{XY,C})$的计算公式如公式(1)所示.

$$z(\rho_{XY,C})=\frac{\sqrt{n-|C|-3}}{2}\cdot\log\left|\frac{1+\rho_{XY,C}}{1-\rho_{XY,C}}\right| \tag{1}$$

其中,n 为混沌实验数据集中性能指标的数量,$|C|$为 C 中性能指标的数量.通过文献[75]中的论证可知,满足标准正态分布.给定阈值 p(本文中 p=0.05),若$|z|<p$,则接受假设$\rho_{XY,C}$=0,说明 X,Y 在给定条件 C 下独立,并删除因果关系图中对应的边 X—Y.

3.3 因果关系边权重赋值

在第 3.2 节得到的性能指标之间的因果关系图中,一个性能指标会存在多条指向该指标的因果关系链路.为了能够将多条因果关系链路排序,优先输出可能性较高的因果关系链路,本阶段为因果关系图中的每一条因果关系边赋予权重值,作为因果关系链路的排序依据.由于韧性风险是基于微服务系统的服务降级上所提出的,且服务性能指标的上升和下降通常由其他系统性能指标的上升和下降引起,本文将混沌实验数据集中各性能指标的上升和下降变化作为因果关系边权重赋予的依据.如果在相近的时间段内两个性能指标同时发生了明显的变化,则说明这两个性能指标很有可能是互相影响的.

借鉴文献[76]中判断一个事件是否对时序数据产生影响的思路,为了检测某一性能指标的上升和下降,本文提出的分析方法将该项性能指标的时间序列数据按照指定的数据量依次划分成若干个数据窗口,对每两个相邻的窗口Γ^{front},通过 t 检验(student t's test) 计算Γ^{front} 和Γ^{rear} 之间的检验统计量 t_{score}.t_{score} 的计算公式如公式(2)所示.

$$t_{score}=\frac{\mu_{\Gamma^{front}}-\mu_{\Gamma^{rear}}}{\sqrt{\frac{\sigma^2_{\Gamma^{front}}+\sigma^2_{\Gamma^{rear}}}{n-1}}} \tag{2}$$

其中,n 为数据窗口的大小,$\mu_{\Gamma^{front}}$,$\mu_{\Gamma^{rear}}$,$\sigma^2_{\Gamma^{front}}$,$\sigma^2_{\Gamma^{rear}}$ 分别为窗口Γ^{front} 和Γ^{rear} 内性能数据的均值和方差.若计算得到某两个窗口的 t_{score} 的绝对值大于某个特定阈值在自由度 n–1 时对应的检验统计量(如在阈值为 0.05、窗口大小为 20 时,t 检验对应的检验统计量为 1.7291),则说明性能指标在两个窗口之间性能发生了显著的上升或下降变化.t_{score} 为正值时,性能指标显著下降;t_{score} 为负值时,性能指标显著上升.图 5 展示了通过上述方法检测一个服务的性能指标变化的样例,在性能数据上标注的绿色点表示识别到的性能上升变化,红色点表示识别到的性能下降变化.

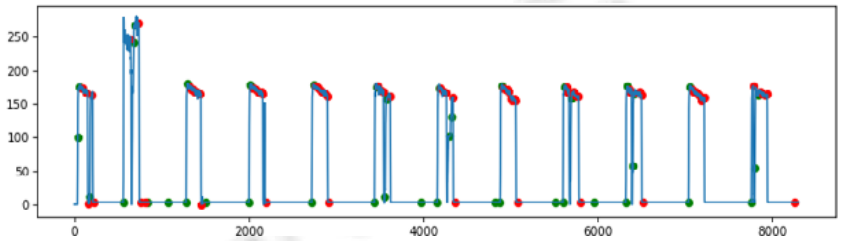


Fig.5 Detected performance changes by student t's test

图 5 使用 t 检验检测到的性能数据变化

通过上述对性能变化检测的方法,一个性能指标的时间数列数据可以转换成图 6 中的性能变化序列,其中,0 表示性能指标没有显著变化,1 表示性能上升,–1 表示性能指标下降.

…00000100000–10000000100000–100000…

Fig.6 Performance change series

图 6 性能变化序列

随后,对于因果关系图中的每一条因果关系边,以边上两个性能指标节点的性能变化序列作为输入,计算两个监控指标之间的皮尔森系数,即可得到性能指标之间的在性能变化上的关联程度.图 7 展示了一条因果关系边上的两个性能指标根据性能上升下降的变化构建性能变化序列,并以性能变化序列之间的关联系数为因果关系边赋予权重的整个过程.

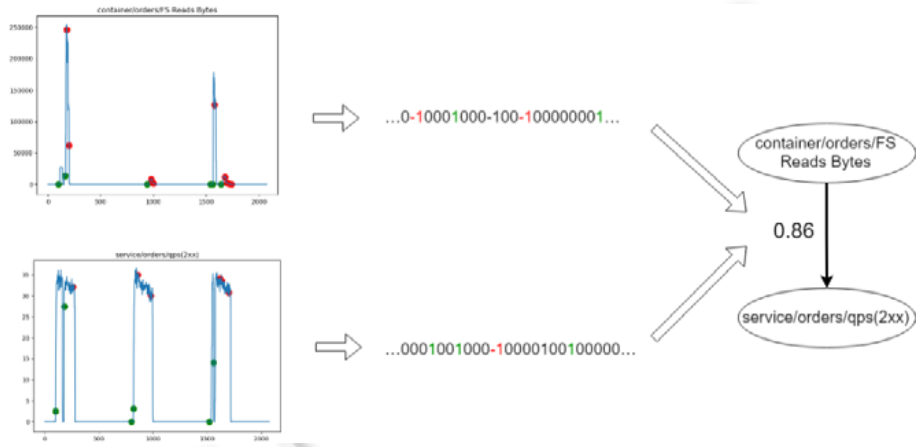


Fig.7 Causality edge weight assignment

图 7 因果关系边权重赋值过程

3.4 韧性风险影响链路输出

基于在第 3.2 节中的因果关系图,韧性风险影响链路将通过以下方法生成:将不满足服务韧性目标的服务性能所对应的节点设为目标节点,在因果关系图中寻找所有直接或间接指向目标节点的性能节点,最后构建出所有由这些节点组成,并最终指向目标节点的因果关系链路.与现有的使用因果关系图的根因诊断的研究相同,本文以深度优先遍历的思路实现了路径构建算法,并最终将所有的因果关系链路作为结果输出.

由于在因果关系图中同时包含了有向边和无向边,在因果关系链路构建的过程中,需要对深度优先算法做出相应的改动,将无向边当作两条方向相反的有向边处理.此外,在因果关系图中进行路径遍历,通常会返回一条以上的因果关系链路.为了对这些因果关系链路进行排序,以第 3.3 节中设置的因果关系边权重为依据,本文提出的分析方法对因果关系链路的构建算法进行了如下改进:当算法遍历到因果关系图中的某一个节点时,算法将优先遍历与该节点相连的权重值较高的边.因此,算法在最后输出韧性风险影响链路的过程中将以可能性从高到低的顺序依次输出可能的因果关系链路,图 8 为一个基于边权重的因果关系链路排序的样例.

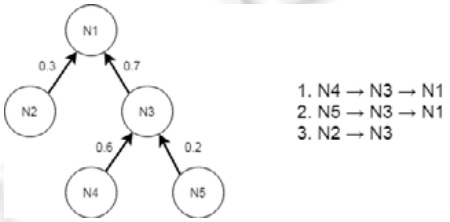


Fig.8 Sort causality chains by weight of causality edges

图 8 根据因果关系边的权重实现因果关系链路的排序

整个因果关系链路的构建算法如算法 2 所示.算法将输出已排序的若干以韧性风险所影响的服务性能指标为最终节点的韧性风险影响链路,运维人员能够按照韧性风险影响链路中的性能指标对韧性风险所带来的影响进行进一步分析.

算法 2. 因果关系链路构建算法.

输入:因果关系图 *G*,起始节点 *N*.

输出:因果关系链路集合 *path_strings*/

```
1.  function searchSource(G,N)
    //在因果关系图 G 中寻找异常性能节点 N 的影响链路
2.      var search_result   //图搜索结果,搜索的结果为一个树结构
3.      search_result.node←N
        search_result.children←[·]
4.      _searchSources(G,N,search_result)   //以 N 作为源节点开始链路搜索
5.  //根据输出结果构建因果链路集合
6.  path_strings←[·];   //待输出的链路集合
7.      path_strings.push(N.name);
        //将只有 N 一个节点的链路作为初始输入的缓存链路
8.      _build_path_strings(search_result.children,path_strings[0]);
        //通过图搜索结果构建韧性风险影响链路
9.      return path_strings;
10. function _searchSource(graph,node,search_result)
    //子算法:在 graph 中搜索影响节点 node 的节点,并把结果存在 search_result 中
11.     links=getLinks(graph,node)
        //获取 graph 中所有节点中包含 node 的边的集合 links
12. if links.length<1:
13. return;
          //node 的邻接节点中没有继续需要追溯的节点,停止继续链路搜索
14.     sub_graph=graph.remove(links);
        //构建一张新的子图,在递归中使用子图进行根因搜索,以保证从两个方向都能追溯的边不会因为在
        之前的递归中已经遍历而不会被再次搜索.
15.     links=links.sort(links.weight)   //根据边的权重排序,权重较高的边将优先输出
16. for link in links:
17. other_node←link 中除 node 外的另一个端点;
18. var child_result   //将 link 的另一个端点当作下一个需要追溯的节点
19. child_reuslt.node←other_node
20. child_result.children←[·]
21. search_result.children.push(child_result);
22. _searchSource(sub_graph,sub_node,search_result.children[sub_node]);
    //以递归的方式继续追溯链路
23. function _build_path_strings(node_children,cur_string)
    //子算法:在缓存路径 cur_string 后根据 children 继续构建链路
24. if node_children.length<1:
25. return;   //已没有后续节点,停止构建链路
```

```
26. path_strings.remove(cur_string)   //在最终输出链路中删除缓存链路
27. for child in node_children:
28. new_string=child.name+"→"+cur_string
        //构建新的缓存链路,由于节点到节点的追溯过程具有方向性,算法在链路分析结果中把无向边当作
        有向边输出
29. path_strings.push(new_string);   //在最终输出中添加缓存链路
30.     _build_path_string(child.children,new_string);
    //递归调用链路生成方法以输出所有可能的链路
```

4 案例研究以及结果分析

为了验证本文提出的微服务韧性风险识别和分析方法的有效性,本文选取了由文献[77]所提出的微服务基线测试套件 Sock-Shop(https://microservices-demo.github.io/)作为实验对象,对本文提出的方法进行案例验证.案例研究中将验证以下研究问题.

(1) 本文提出的韧性风险识别方法是否能够自动地识别出目标系统的韧性风险?

(2) 本文提出的韧性风险分析方法是否能够有效地分析出目标系统在发生韧性风险时因果的影响链路?

4.1 目标系统以及实验环境

图 9 为 Sock-Shop 的系统架构图.整个 Sock-Shop 系统由前端(front-end)、订单服务(order)、支付服务(payment)、用户服务(user)、商品详情服务(catalogue)、购物车服务(cart)以及邮寄服务(shipping)这几个微服务组成,每一个微服务均可以独立运行且拥有独立的数据库.为了保证微服务技术异构性[10]的特点,Sock-Shop 中不同微服务的实现语言以及使用的数据库各不相同.Sock-Shop 中,微服务之间的通信以 HTTP 通信为主,并使用了 RESTful 的接口设计风格.

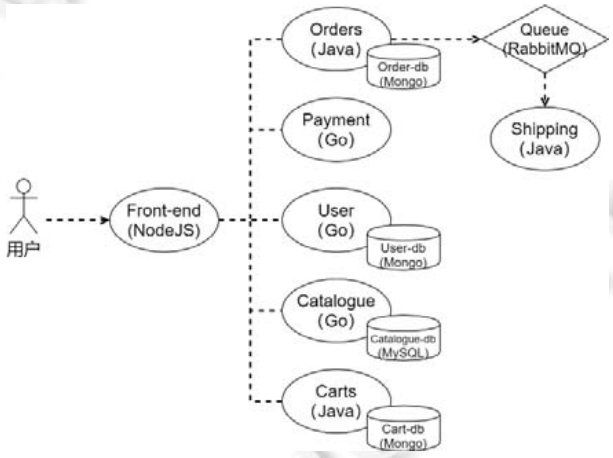


Fig.9 Sock-Shop architecture

图 9 Sock-Shop 系统架构

图 10 为案例研究中 Sock-Shop 的部署环境图,整个部署环境将由一个控制服务器和由一个 Master 节点以及 3 个 Worker 节点所构建的容器集群管理平台 Kubernetes(https://kubernetes.io/)组成.Sock-Shop 系统被部署在 Kubernetes 集群上,Sock-Shop 中的每一个微服务均在一个或多个容器上部署.控制服务器负责韧性风险识别方法的自动化实现.其中,Sock-Shop 系统的自动部署和混沌实验的进行由部署在控制服务器上的 Jenkins(https://jenkins.io/)容器完成,压力测试框架 Locust(https://www.locust.io/)负责 Sock-Shop 应用压力场景的模拟,混沌测试工具 Chaos-Toolkit(https://chaostoolkit.org/)负责系统环境扰动的实现.部署环境的各个服务器配置相同,具体

配置见表 1.部署环境中的容器均按默认配置部署,并未对各容器的资源分配情况进行额外设置,因此,Worker 节点上运行的各容器资源分配模式均采用容器默认的按需分配方式.

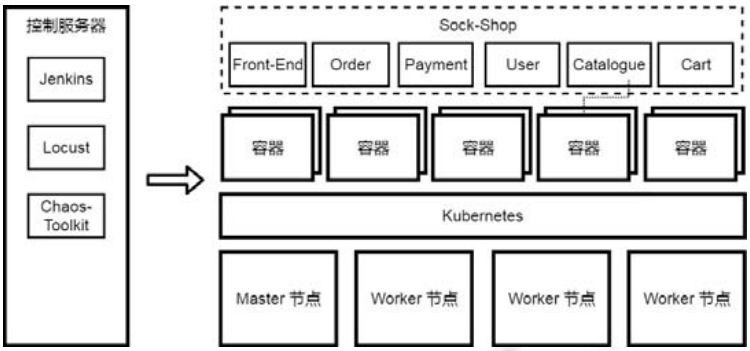


Fig.10 Sock-Shop system deployment environment

图 10 Sock-Shop 的部署环境

Table 1 Server configuration of the deployment environment

表 1 部署环境的服务器配置

配置	参数
操作系统	Ubuntu Server 16.04
CPU	Intel Xeon E5-2620*2
内存	64GB ddr4
磁盘	3TB
网卡	1 000Mb/s

上述目标系统和实验环境满足了本文第 2.1 节中所提出的各项系统前提:自动化部署组件 Jenkins 与 Sock-Shop 本身提供的 Kubernetes 部署配置文件满足了条件(1)的系统自动化构建能力;Sock-Shop 系统已处于可发布运行的状态因此满足条件(2);Kubernetes 本身集成的时序数据收集工具 Prometheus 使目标系统满足了条件(3).案例研究将首先按照条件(4)、条件(5)提出的假设下对目标系统进行韧性风险识别.

4.2 Sock-Shop的韧性风险识别

为了验证研究问题 1,本案例研究中将根据本文第 2 节所提出的韧性风险识别方法寻找 Sock-Shop 系统中的韧性风险.

4.2.1 韧性风险识别

在 Sock-Shop 中,所有的用户操作均为网页端的点击操作,因此,Sock-Shop 的系统服务均为事务性的 HTTP 服务.在本案例研究中,服务平均响应时间(response time)以及响应成功率(success rate)被作为所有服务的基础性能指标.Sock-Shop 的核心服务的服务性能指标将根据服务的业务关注点设立.表 2 展示了在案例研究中针对 Sock-Shop 系统各服务所设计的服务性能指标.

Table 2 Service performance indicators of Sock-Shop

表 2 Sock-Shop 的服务性能指标

服务	服务性能指标名称	描述
全部服务	响应时间	从用户请求发出到收到系统回复花费的时间
–	响应成功率	单位时间服务返回正常系统回复的成功率
订单服务	每秒成功交易量	每秒成功完成订单的数量
商品信息服务	每秒执行事务数量	服务每秒处理的请求数量
用户登录服务	登录成功率	单位时间内所有用户登录请求中登录成功的比率

案例研究中使用的部署环境是实验环境而非实际生产环境,且Sock-Shop本身为开源系统并未对性能和容错方面有严格的设计.通过观察在实验环境中引入各种系统环境扰动后环境的服务降级情况,并参考ETSI标准中韧性测试的建议值[78],对 Sock-Shop 中所有的系统服务的响应时间以及响应成功率这两项服务性能指标的基线值的设置也相对实际生产环境适当放宽,分别为5s以及90%.通过对目标系统的压力测试发现:在没有引入系统环境扰动的情况下,目标系统每分钟能最多能接受约 200 个用户请求.因此,以每分钟 200 个请求的系统峰值和Sock-Shop整体的业务流程为依据,案例研究中设计了表3中的系统压力场景,并把在这些压力场景下收集到的系统服务性能数据作为系统性能基线.压力场景中,对用户操作的模拟是通过编写脚本访问对应 HTTP 接口实现.用户行为模拟脚本会被压力测试工具Locust在每分钟并行地触发,以实现系统环境压力的模拟.

Table 3 Workload scenarios of Sock-Shop

表 3 Sock-Shop 的压力场景

压力场景	压力场景说明
普通场景	每分钟有 50 个用户完成登录→查看商品→添加商品至购物车→提交订单的过程.其中,浏览以及添加至购物车的商品数量为 1~5 之间的一个随机值
普通场景(高压力)	每分钟有 100 个用户完成登录→查看商品→添加商品至购物车→提交订单的过程.其中,浏览以及添加至购物车的商品数量为 1~5 之间的一个随机值
促销活动	每分钟有 200 个用户提交订单的请求
促销活动前期	每分钟有 200 个用户完成登录→查看商品→添加商品至购物车的过程

在对表2中的Sock-Shop各服务性能指标设立韧性目标的过程中,平均响应时间和响应成功率这两项通用服务性能指标的韧性目标设立,包含性能降级程度以及降级恢复时间这两个维度.对于不同服务各自的服务性能指标,服务韧性目标的维度选取将考虑到各个服务各自的业务特点.表4为Sock-Shop各服务性能指标的具体服务韧性目标.为了分析混沌实验次数与识别的韧性风险数量之间的关系,案例研究中,对目标系统分别以混沌实验次数 10 次、20 次、50 次、100 次进行韧性风险识别过程.考虑到每次混沌实验中 Sock-Shop 的自动构建、部署过程和自动销毁过程,每次混沌实验的实验最大时长被设置为 30 分钟.对 Sock-Shop 混沌实验中的系统压力场景的模拟将随机选择表 3 中的一个压力场景.在混沌实验中引入的系统环境扰动所涉及的系统资源以及系统资源对应的系统环境扰动事件见表 5.

Table 4 Service resilience goals of Sock-Shop

表 4 Sock-Shop 各服务的韧性目标

服务	服务性能指标	性能降级程度	降级恢复时间	降级损失
全部服务	响应时间	10s	5s	–
–	响应成功率	20%	5s	–
订单服务	每秒成功交易量	–	–	500 交易
商品信息服务	每秒执行事务数量	–	5s	–
用户登录服务	登录成功率	20%	–	–

Table 5 System environment disruptions events of different system resource types

表 5 各个系统资源类型的系统环境扰动事件

系统资源类型	系统环境扰动事件
CPU	CPU 资源占用至 100%
内存	内存占用至 100%
磁盘	磁盘 I/O 阻塞
网络	包传递延时/包损坏
进程	进程被杀死
容器	容器被关闭
服务器	服务器宕机

混沌实验过程中各个服务的服务性能数据将通过两个方式收集:① 对于响应时间和响应成功率等通用的服务性能数据,使用监控工具 Prometheus(https://prometheus.io/)组件进行数据收集;② 对于各个服务中针对业务的服务性能数据,利用系统压力模拟工具在压力模拟的同时收集各个请求的返回结果,对返回结果进行统计

得到实时的服务性能数据.在收集服务性能数据的同时,系统其他各项系统资源的性能数据也将通过 Heapster (https://github.com/kubernetes/heapster),Zabbix(https://www.zabbix.com/)等工具采集,用于识别出韧性风险后对韧性风险与服务降级的因果关系分析.

4.2.2 实验结果分析

案例分析中,对目标系统分别以混沌实验次数 10 次、20 次、50 次、100 次进行了韧性风险识别.由于每次引入的系统环境扰动都是随机生成的,为了观察随机性对识别出的韧性风险数量的影响,各混沌实验次数的韧性识别过程均实行了 3 次.表 6 展示了不同混沌实验次数下所识别出的韧性风险数量.

Table 6 Number of identified resilience risks in different chaos experiments

表 6 不同混沌实验次数下识别的韧性风险数量

混沌实验次数	第 1 次识别数量	第 2 次识别数量	第 3 次识别数量
10	0	2	0
20	2	5	3
50	5	10	18
100	20	12	24

从表 6 的验证结果可以看出,本文提出的微服务韧性需求识别方法能够识别出一定数量的目标微服务架构系统的韧性风险.虽然在相同混沌实验次数下识别到的韧性风险数量有一定差异,但是低混沌实验次数识别到的韧性风险数量均低于高次数混沌实验识别到的数量,说明识别过程中混沌实验的次数越多,通过混沌实验发现的韧性风险越多.从相同混沌实验次数的风险识别结果比较中可以看出:系统扰动生成的随机性对识别出的韧性风险数量有着较大影响,在混沌实验次数较少的情况下,有着没有识别出任何韧性风险的可能性.

由于混沌实验中的系统环境扰动是随机生成的,为了验证韧性风险识别过程中重复进行相同混沌实验的可能性,我们对每一次识别过程中生成的系统环境扰动以及韧性风险进行了统计,统计结果见表 7.从表 7 中可以看出:虽然在混沌实验次数较大的情况下出现了重复的系统环境扰动,重复生成的系统环境扰动数量相对整个识别过程中生成的系统环境扰动数量可以忽略不计.

Table 7 Number of duplicate system disruptions and resilience risks in different chaos experiments

表 7 混沌实验中重复的系统环境扰动和韧性风险数量统计

混沌实验次数	重复生成的系统环境扰动数量(3 次识别过程总和)	重复识别的韧性风险数量(3 次识别过程总和)
10	0	0
20	0	0
50	1	0
100	3	0

4.3 Sock-Shop系统的韧性风险分析

限于文章篇幅,本文将以一个实际的韧性风险为例,展示对 Sock Shop 系统韧性风险的分析过程.图 11 为一次混沌实验中订单服务每秒成功交易量的性能曲线,在混沌实验中引入的系统环境扰动为对 carts 服务的容器的 CPU 占用率提升至 100%.图 11 中,蓝色曲线为混沌实验中收集到的性能曲线,红色曲线为通过 Holt Winters 算法以正常压力场景下的历史数据为输入得到的性能基线曲线.可以看出,图 11 中一段时间内订单服务的实际性能明显低于其性能基线,说明订单服务由于环境扰动进入了服务降级状态,且通过计算可得知该次服务降级的性能损失高于其性能损失阈值(500 个交易).因此,本次混沌实验中引入的系统环境扰动被认定为韧性风险.

在混沌实验的过程中,通过 Prometheus 组件实际采集到的各服务的性能指标包括服务的每秒请求数量(*qps*,其中包括成功的请求量 *qps*(2*xx*)以及失败的请求量 *qps*(4*xx*))以及服务请求延时这两项性能指标.在韧性风险识别阶段中的设立服务性能指标(表 1)将通过以上指标换算得到.如,

$$\text{响应成功率}=qps(2xx)/(qps(2xx)+qps(4xx))).$$

除了收集服务性能的监控数据外,同时会收集容器以及节点的性能数据,具体收集的性能数据见表 8.

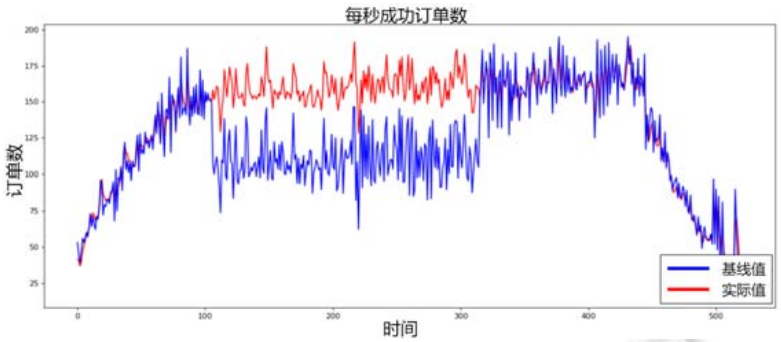


Fig.11 A service degradation occurred in a chaos experiment

图 11 一次混沌实验中发生的服务降级

Table 8 Collected performance data in chaos experiments

表 8 混沌实验过程中收集的性能数据

系统资源类型	性能指标	说明
服务	*qps*(2*xx*)	每秒成功请求量(requests/s)
	qps(4*xx*)	每秒失败请求量(requests/s)
	Latency	服务响应延时(s)
容器	CPU usage	CPU 使用率(%)
	MEM usage	内存使用率(%)
	FS reads bytes	每秒文件系统读入字节数(bytes/s)
	FS write bytes	每秒文件系统写入字节数(bytes/s)
	Network input packets	每秒网络传入包数(pakcets/s)
	Network output packets	每秒网络传输包数(pakcets/s)
节点	CPU usage	CPU 使用率(%)
	MEM usage	内存使用率(%)
	Disk reads bytes	每秒磁盘读入字节数(bytes/s)
	Disk write bytes	每秒磁盘写入字节数(bytes/s)
	Network input packets	每秒网络传入包数(pakcets/s)
	Network output packets	每秒网络传输包数(pakcets/s)

在性能指标的因果关系图构建过程中,需要保证每个时间戳下各性能指标均存在有效值,但是表 3 中各项通过换算得到的服务性能指标并不能保证这一点(如在 *qps*(2*xx*)和 *qps*(4*xx*)均为 0 的情况下,相应的成功率是无效值),因此在韧性风险的分析过程中,直接使用了实际收集的服务性能指标进行韧性风险分析,构建韧性风险影响链路时的目标节点将使用与违反服务韧性目标的性能指标直接相关的服务性能指标(如订单服务的每秒成功交易量与其每秒成功请求数直接相关).通过因果搜索算法得到各项性能指标之间的因果关系图如图 12 所示(电子版文档中,可以放大该图查看细节).其中,蓝色节点为服务的性能指标,红色节点为容器的性能指标,绿色节点为 kubernetes 集群的工作节点的性能指标.

图 13 为图 12 中所有服务每秒请求数量(*qps*)的因果关系图,在与图 9 的比较后可以看出:仅通过性能数据所构建的因果关系图基本与 sock-shop 的系统架构图一致,并且能够反映出各微服务在业务场景上的先后关系(如浏览商品(catalogue 服务)→添加购物车(carts 服务)→用户下单(orders 服务)).说明通过因果搜索算法生成的因果关系图在实际物理关系上具有一定程度的准确性.

以图 12 的因果关系图为输入,计算图中所有因果关系边的权重,并以服务降级的性能指标(每秒成功交易量)直接相关的服务性能指标(service/orders/*qps*(2*xx*))为终点,按照深度优先遍历的方式寻找所有直接或间接指向该节点的性能指标.最终得到的影响服务降级的性能指标以及对应的因果关系边的权重如图 14 所示.

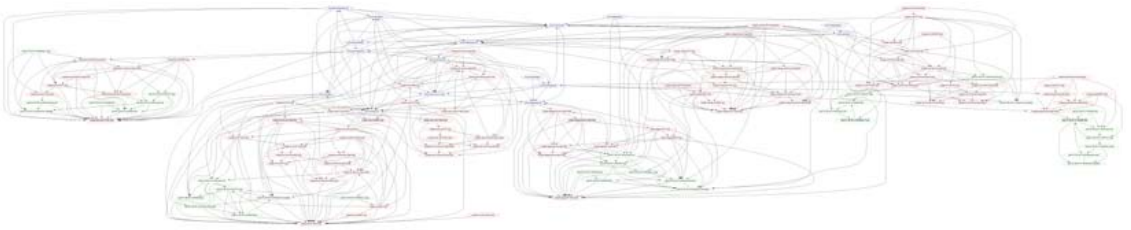


Fig.12 Causality graph of performance indicators of all performance indicators

图 12 所有性能指标之间的因果关系图

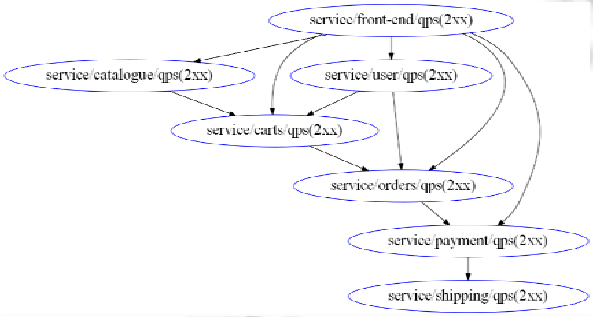
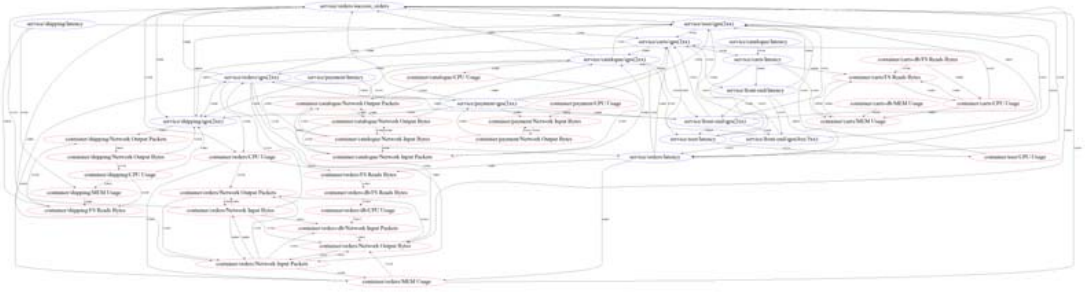


Fig.13 Causality relations among service QPS

图 13 服务 QPS 之间的因果关系

(a) 图为部分性能指标之间的因果关系及权重



(b) 图为全部性能指标之间的因果关系及权重

Fig.14 Performance indicators that influence the service degradation and corresponding causality edges

图 14 影响服务降级的性能指标以及对应的因果关系边

将图 14 中的因果链路根据因果关系边的权重进行排序,最终输出的韧性风险影响链路如图 15 所示.图 15 中标注为红色的性能指标(container/carts/CPU Usage)被混沌实验中引入的环境扰动直接产生影响的性能指标,可以看出,优先输出的韧性风险影响链路均命中了引入的系统环境扰动.

1. container/carts-db/FS Reads Bytes→container/carts/CPU Usage→service/carts/*qps*(2*xx*)→service/carts/latency→service/front-end/latency→service/front-end/*qps*(2*xx*)→service/front-end/*qps*(4*xx*/5*xx*)→service/orders/latency→service/catalogue/*qps*(2*xx*)→service/orders/*qps*(2*xx*)→container/orders/CPU Usage→container/orders/Network Output Packets→container/orders/Network Input Bytes→container/orders-db/Network Input Packets→container/orders/Network Output Bytes→container/orders/Network Input Packets→container/orders/MEM Usage→service/orders/*qps*(2*xx*)
2. container/carts-db/FS Reads Bytes→container/carts/FS Reads Bytes→container/carts/CPU Usage→service/carts/*qps*(2*xx*)→service/carts/latency→service/front-end/latency→service/front-end/*qps*(2*xx*)→service/front-end/*qps*(4*xx*/5*xx*)→service/orders/latency→service/catalogue/*qps*(2*xx*)→service/orders/*qps*(2*xx*)→container/orders/CPU Usage→container/orders/Network Output Packets→container/orders/Network Input Bytes→container/orders-db/Network Input Packets→container/orders/Network Output Bytes→container/orders/Network Input Packets→container/orders/MEM Usage→service/orders/*qps*(2*xx*)

…

Fig.15 Resilience risk influence chain

图 15 韧性风险影响链路

通过观察图 15 韧性风险影响链路中各性能指标的变化,发现对 cart 容器进行 CPU 加压(container/cart/cpu)提高了 cart 服务的延时(service/cart/latency),并减少了其他服务的请求量(service/服务/*qps*(200)),因此减少了 order 容器的网络请求传入量(container/orders/Network Input Packets)以及 order 容器内存中缓存的订单数量(container/orders/MEM Usage),最终影响交易成功的订单数量(service/orders/*qps*(2*xx*)).图 16 展示了上述性能指标在引入扰动时的具体性能曲线.从上述分析中可以看出,案例研究中得到的因果关系链路一定程度上体现了从扰动到服务降级的传播过程.

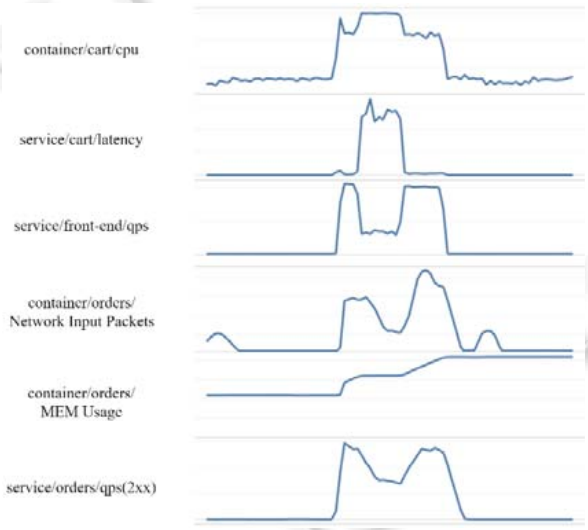


Fig.16 Performance curves of different performance indicators when a disruption was injected

图 16 引入系统环境扰动时各性能指标的性能曲线

5 进一步分析与总结

5.1 实验结果的进一步分析

为了进一步验证本文提出的韧性风险的分析方法是否能够有效地分析出目标系统在发生韧性风险时因果的影响链路,本文分析了案例研究中的多次混沌实验中收集到的系统性能数据,以评估韧性风险影响链路的准

确度.本文中使用 precision(查准率)和 recall(查全率)作为评估韧性风险影响链路准确度的度量指标.由于本文提出的分析方法最终输出排序后的多条链路,precision 和 recall 中 TP 的判断标准为输出的前 k 条链路是否命中被引入的环境扰动,其对应的 precision 和 recall 值分别记作 $precision_k$ 和 $recall_k$.FP(false positive)表示分析方法输出的前 k 条链路中没有命中环境扰动;FN(false negative)表示分析方法输出前 k 条链路中没有命中环境扰动,但是剩余的链路中命中了环境扰动.为了方便统计韧性风险影响链路是否命中了被引入的环境扰动,本文中仅分析了直接与系统性能有关的环境扰动所对应的混沌实验数据集(环境扰动包括:将 CPU 利用率提升至 100%(CPUBurnout)、将内存占用率提示至 100%(Mem overload)、磁盘 IO 阻塞(disk I/O block)以及网络传输延时(network Jam)).

- 通过因果关系搜索算法所构建的因果关系图是否准确?

在第 4.3 节的实验中,已证明了通过因果关系算法构建的性能指标因果关系图与 Sock-Shop 的实际服务调用图基本一致.为模拟验证在服务数较多、调用关系较深情况下的因果搜索算法构建的因果关系图的准确性,本文通过因果关系模拟工具 tetrad(http://www.phil.cmu.edu/projects/tetrad/)构建出包含不同数量节点和因果关系边的因果关系图,并在假定的因果关系图上模拟大量数据.因果关系模拟算法将在模拟的数据中寻找变量之间的因果关系,并将得到的因果关系与假定的因果关系进行比较,统计实际不存在的、存在但未识别的以及方向识别错误的因果关系边数,以验证算法分析得到的因果关系的准确性.

表 9 展示了在不同节点和因果关系边数量下,因果关系搜索算法的准确度.从表 9 可以看出:因果搜索算法能够识别出大部分的因果关系,与实际因果关系的准确度基本高于 90%.对于实际大规模微服务架构系统的验证,将在本文的后续研究工作中进一步展开.

Table 9 Accuracy of the causality search algorithm under different number of nodes and edges

表 9 不同节点和边数量下因果关系搜索算法的准确度

节点数	边数	实际不存在的因果关系边数	未识别的因果关系边数	方向错误的因果关系边数	Precision	Recall
10	20	1	0	0	0.95	1
10	30	0	0	0	1	1
15	30	2	5	0	0.92	0.83
15	45	3	8	0	0.93	0.82
20	40	3	10	0	0.91	0.75
20	60	5	9	0	0.91	0.85
30	60	10	12	1	0.81	0.78
30	90	7	11	0	0.92	0.88
50	100	9	15	2	0.88	0.83
50	150	10	12	1	0.92	0.91

- 针对服务韧性设计的因果关系边权重赋值方法是否能够优化对韧性风险的分析过程?

本文中比较了本文提出的因果关系边权重赋值方法和现有文献中使用的因果关系边权重赋值方法所得到的因果关系链路的准确度,以验证本文提出的针对韧性风险的因果关系边权重赋值方法的有效性,被比较的权重赋值方法有:

(1) 3σ检测:通过 3σ规则判断系统性能是否发生了异常,并将系统性能的时间序列数据转换成仅包含 0 和 1 的性能异常序列.随后,以性能异常序列计算性能指标之间的皮尔森系统作为因果关系边的权重.该方法为 Microscope[64]中使用的方法.

(2) 阈值关联:为了与 CauseInfer[65]中对离散系统性能指标的因果关系分析方法比较,在本案例研究中,将为每一类型的系统性能指标设立若干的阈值,并根据阈值将连续的性能指标序列转换成离散序列.随后,同样使用皮尔森系数为因果关系边构建权重.

(3) 直接关联:按照 CloudRanger[63]中的方法,直接使用两个性能指标数据的皮尔森系数作为因果关系边的权重.

各因果关系边权重赋值方法在不同类型的环境扰动下对最终输出的因果关系链的准确度的影响见表 10.从表 10 中可以看出,本文提出的针对系统性能变化的因果关系边权重的赋值方法在分析韧性风险上相对于其

他权值赋予方法有更高的准确性.在 4 种引入的环境扰动中,由于容器网络相关的性能指标曲线与服务性能的曲线有较高的相似性,识别网络阻塞的准确率相对于其他类型系统扰动更高.在表 10 的实验结果中,基于 3σ检测的韧性风险影响链路的准确度低于参考文献[64]中故障根因分析的准确度.这是因为基于 3σ的异常检测方法更适用于寻找时间序列数据中的明显的离群点,而在服务降级发生的过程中,系统性能会较长时间处于性能较低的状态,无法被识别为异常.

Table 10 Accuracy of causality edge weight assignment methods under different environment disruptions

表 10 不同类型系统环境扰动下各因果关系边权重赋值方法的准确率

赋值方法	本文提出的方法		3σ检测(MicroScope)		阈值(CauseInfer)		直接关联(CloudRanger)	
故障类型	准确度							
	$precision_1$	$recall_1$	$precision_1$	$recall_1$	$precision_1$	$recall_1$	$precision_1$	$recall_1$
CPUBurnout	**82.38**	**86.20**	39.98	42.00	77.53	81.82	67.87	74.31
MEM overload	**89.37**	**91.23**	43.48	44.34	84.56	86.15	79.88	83.22
Disk I/O block	**89.39**	**92.58**	43.58	45.52	84.96	89.49	79.91	85.72
Network Jam	**94.45**	**97.22**	46.53	48.11	91.67	95.24	89.19	84.52

- 因果关系链路是否能够命中注入的系统环境扰动?

本文中对目标系统的各个服务多次注入了各种类型的系统环境扰动(为了保证各服务和各扰动类型的实验次数,引入的环境扰动不一定为韧性风险),并验证本文提出的分析方法输出的韧性风险影响链路的准确度.表 11 为各个服务在不同环境扰动下的得到的影响链路分别在 k=1 和 k=2 情况下的 *precision* 和 *recall* 值.从表 11 中可以看出:通过本文提出的韧性风险分析方法对于识别案例研究中所注的各类型环境扰动均有较高的准确度,且得到的韧性风险影响链路在大部分服务下能够保证 80%以上的准确率,仅对 Shipping 服务施加环境扰动时链路的准确率相对于其他服务较低.这是由于 Shipping 服务仅会被 Order 服务创建完用户订单后被调用,且不会调用其他服务.故 shipping 服务的响应时间及成功率不会影响到其他服务,较难产生明显的因果关系.

Table 11 Accuracy of the resilience risk analysis method under different services and disruptions

表 11 各服务在不同类型环境扰动下韧性风险分析方法的准确率

	Front-end	Catalogue	User	Carts	Orders	Shipping	Payment
CPUBurnout							
$precision_1$	100	81.82	91.67	80	92.31	33.33	63.64
$precision_2$	100	81.82	91.67	90	92.31	33.33	63.64
$recall_1$	100	90	91.67	80	100	33.33	77.78
$recall_2$	100	90	91.67	90	100	33.33	77.78
MEM Overload							
$precision_1$	100	91.67	100	88.89	100	36.36	75
$precision_2$	100	92.31	100	88.89	100	36.36	75
$recall_1$	100	91.67	100	100	100	36.36	75
$recall_2$	100	92.31	100	100	100	36.36	75
Disk I/O Block							
$precision_1$	100	92.31	100	85.71	100	50	66.67
$precision_2$	100	100	100	85.71	100	50	66.67
$recall_1$	100	92.31	100	92.31	100	60	81.82
$recall_2$	100	100	100	92.31	100	60	81.82
Network Jam							
$precision_1$	100	100	100	91.67	100	66.67	83.33
$precision_2$	100	100	100	91.67	100	66.67	83.33
$recall_1$	100	100	100	100	100	66.67	100
$recall_2$	100	100	100	100	100	66.67	100

实验结果中,有几次环境扰动并没有被准确地识别.通过对这几次实验的服务性能曲线及分析过程中得到的因果关系图的分析,发现这几次混沌实验在引入环境扰动后并没有产生明显的服务降级,造成在因果关系图中引入扰动的性能指标节点的因果关系边权重小于其他性能指标节点的因果关系边权重,故在构建的因果关系链路中排序较后没有输出.没有产生严重服务降级的系统扰动将不会被识别为韧性风险并进一步分析,因此,实验中根因定位失败的几次环境扰动不会影响本文提出的韧性风险识别和分析方法.

- 混沌实验数据集中性能指标的数量以及数据条数是否会对韧性风险分析方法产生影响?

为了分析混沌实验数据集中性能指标的数量及数据条数对韧性风险分析方法的影响,实验中分别统计了在较少性能指标(20 个)情况下和以 50 个性能指标为增量,在 50/100/150 个性能指标情况下的系统资源消耗,并分别以 25/50/250 的梯度递增地统计了不同数据条数下韧性风险分析方法的系统资源消耗.表 12 为不同性能指标和数据条数下韧性风险分析方法对系统上的资源消耗,从表 12 中可以看出:本文提出的韧性风险分析方法能够在较短的时间内分析出混沌实验数据集中的因果关系链,且消耗较少的 CPU 和内存资源.相对于数据条数,混沌实验数据集中的性能指标数量对分析所花费时间有着更明显的影响.如何在保证算法准确度的同时降低计算时间,将是本文的后续研究目标之一.

Table 12 System resource consumption of the resilience risk analysis method under different data length

表 12 不同数据量下韧性风险分析方法的系统资源消耗

性能指标数量	20			50			100			150		
数据条数	资源消耗											
	Time (s)	CPU (%)	MEM (MB)	Time (s)	CPU (%)	MEM (MB)	Time (s)	CPU (%)	MEM (MB)	Time (s)	CPU (%)	MEM (MB)
25	4.36	7.92	17.92	10.46	7.94	34.19	15.18	7.39	33.86	34.01	9.56	37.12
50	4.38	7.72	18.71	12.55	8.89	36.65	25.41	7.36	35.77	35.52	8.41	36.00
100	4.50	7.49	18.93	17.42	7.42	35.47	32.61	7.49	37.68	44.05	7.26	38.47
150	4.36	8.99	19.17	16.71	7.53	39.45	31.07	8.72	38.51	46.52	7.23	41.46
200	4.49	7.02	19.06	17.79	7.45	38.09	32.71	7.92	40.56	46.85	8.48	42.99
250	4.38	7.67	19.66	18.81	8.09	37.90	35.74	8.95	39.73	50.42	8.12	45.59
500	4.5	7.44	19.68	20.11	8.92	39.90	35.69	8.48	41.43	53.31	8.02	46.06
750	4.40	7.99	20.12	20.43	8.01	39.84	37.88	8.43	44.33	55.03	7.98	46.09
1 000	4.40	7.13	21.96	20.53	8.37	40.94	40.95	7.21	43.40	55.19	8.58	45.02
1 500	4.43	7.23	21.89	21.05	8.16	44.23	43.39	8.78	44.99	77.79	7.34	46.29
2 000	4.45	7.37	22.07	27.19	8.85	42.27	46.09	8.45	46.83	67.01	8.47	45.07

图 17 为不同数据条数下,韧性风险影响链路的 *Precision* 和 *Recall* 值(k=1).当混沌实验数据集的数据量较小时,由于在因果关系图的构建过程中识别到的因果关系边数量变少,导致了链路的准确度降低;而当数据集的数据条数达到一定数量(本案例中为 200 条数据)后,链路的准确度能保持在 80%以上并处于一个相对稳定的状态.由此可见,本文提出的韧性风险分析方法仅需相对较少的数据条数即可保证算法的准确度.

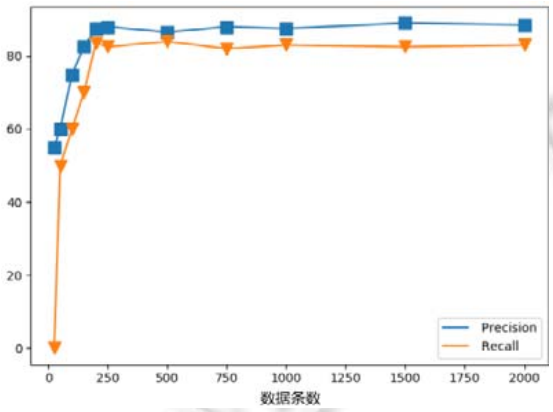


Fig.17 Influence of data length on resilience risk analysis accuracy

图 17 数据条数对韧性风险分析方法准确度的影响

5.2 总 结

本文着手于解决微服务架构系统中韧性风险的度量、识别以及分析问题.基于 MRMM 模型,并结合混沌工程的实践方法,本文提出了针对微服务架构系统的韧性风险识别方法,以识别出严重影响系统服务性能的韧性风险,大幅地减少了韧性风险识别过程中的人力与时间成本.对识别出的韧性风险,本文提出了基于因果搜索算

法的分析方法,最终得到由系统性能指标构成的韧性风险影响链路以供运维人员参考.最后,本文在开源微服务系统 Sock-Shop 上的案例研究也证明了本文提出的韧性风险识别和分析方法能够在微服务架构系统中识别出潜在的韧性风险,并得到具有一定准确性的韧性风险影响链路.

本文需要进一步开展相关研究工作包括以下几个方面.

(1) 混沌实验设计的优化.目前,在混沌实验中引入系统环境扰动纯粹以随机的方式生成.而在本文的案例研究中可以看出,这种方式生成的系统环境扰动是韧性风险的概率并不是很高.如何利用前一次系统迭代中混沌实验的结果以及系统实际运行过程中发现的韧性风险,改进混沌实验中系统环境扰动的生成方式以提高发现韧性风险的概率?如何有效地组合多种不同的系统环境扰动,观察多种扰动对微服务架构系统的共同作用?均是本文后续研究中着手解决的问题.

(2) 韧性风险分析方法的优化.本文中分析得到的因果关系分析链路是以系统性能指标的上升下降变化作为分析基础.而在实际的系统服务降级过程中,异常事件、人为操作等均会导致系统性能指标的变化.如何有效地将异常检测、系统日志挖掘等技术融入韧性风险分析过程中,是本文在提出韧性风险分析方法之后的研究目标.此外,现已有一些文献[79]使用系统挖掘工具(如 sysdig),在避免额外开发的情况下捕获服务之间的调用关系,如何在微服务架构系统中挖掘服务调用关系并利用其优化韧性风险分析方法的分析效率和准确度,也是本文待研究的问题之一.

References:

[1] Lewis J, Fowler M. Microservices: A definition of this new architectural term. 2014. https://martinfowler.com/articles/microservices.html

[2] Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software, 2016,33(3):42−52. [doi: 10.1109/MS.2016.64]

[3] Mauro T. Adopting microservices at netflix: Lessons for architectural design. 2015. https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/

[4] Ihde S. InfoQ—From a monolith to microservices + REST: The evolution of LinkedIn's service architecture. 2015. https://www.infoq.com/presentations/linkedin-microservices-urn/

[5] Calçado P. Building products at soundcloud—Part III: Microservices in scala and finagle. SoundCloud Limited, 2014. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle

[6] Dragoni N, Giallorenzo S, Lafuente AL, *et al*. Microservices: Yesterday, today, and tomorrow. In: Proc. of the Present and Ulterior Software Engineering. Cham: Springer-Verlag, 2017. 195−216. [doi: 10.1007/978-3-319-67425-4_12]

[7] Gunawi HS, Hao M, Suminto RO, *et al*. Why does the cloud stop computing? Lessons from hundreds of service outages. In: Proc. of the 7th ACM Symp. on Cloud Computing. New York: ACM, 2016. 1−16. [doi: 10.1145/2987550.2987583]

[8] ISO/IEC 25010: 2011, Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. Int'l Standards Organization, 2011. https://www.iso.org/standard/35733.html

[9] Gunawi HS, Hao M, Leesatapornwongsa T, *et al*. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In: Proc. of the ACM Symp. on Cloud Computing. New York: ACM, 2014. 7:1−7:14. [doi: 10.1145/2670979.2670986]

[10] Newman S. Building Microservices: Designing Fine-grained Systems. New York: O'Reilly Media, Inc., 2015.

[11] Nadareishvili I, Mitra R, McLarty M, *et al*. Microservice Architecture: Aligning Principles, Practices, and Culture. New York: O'Reilly Media, Inc., 2016.

[12] Nygard MT. Release It! Design and Deploy Production-ready Software. 2nd ed., United States: Pragmatic Bookshelf, 2018.

[13] Windle G, Bennett KM, Noyes J. A methodological review of resilience measurement scales. Health and quality of life outcomes, 2011,9(1):Article No.8. [doi: 10.1186/1477-7525-9-8]

[14] Yin K, Du Q, Wang W, *et al*. On representing resilience requirements of microservice architecture systems. arXiv Preprint arXiv: 1909.13096, 2019.

[15] Boehm B. Software risk management. In: Proc. of the European Software Engineering Conf. Berlin, Heidelberg: Springer-Verlag, 1989. 1−19. [doi: 10.1007/3-540-51635-2_29]

[16] Holling CS. Resilience and stability of ecological systems. Annual Review of Ecology and Systematics, 1973,4(1):1−23. [doi: 10.1146/annurev.es.04.110173.000245]

[17] Hosseini S, Barker K, Ramirez-Marquez JE. A review of definitions and measures of system resilience. Reliability Engineering & System Safety, 2016,145(2016):47−61. [doi: 10.1016/j.ress.2015.08.006]

[18] Xue X, Wang L, Yang RJ. Exploring the science of resilience: Critical review and bibliometric analysis. Natural Hazards, 2018, 90(1):477−510. [doi: 10.1007/s11069-017-3040-y]

[19] Laprie JC. Dependability: Basic Concepts and Terminology. Vienna: Springer-Verlag, 1992. [doi: 10.1007/978-3-7091-9170-5]

[20] Laprie JC. From dependability to resilience. In: Proc. of the 38th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Los Alamitos: IEEE Computer Society, 2008. G8−G9.

[21] Wolter K, Avritzer A, Vieira M, Van Moorsel A, eds. Resilience Assessment and Evaluation of Computing Systems. Berlin, London: Springer-Verlag, 2012. [doi: 10.1007/978-3-642-29032-9]

[22] Trivedi KS, Kim DS, Ghosh R. Resilience in computer systems and networks. In: Proc. of the 2009 Int'l Conf. on Computer-aided Design. New York: ACM, 2009. 74−77. [doi: 10.1145/1687399.1687415]

[23] Bishop M, Carvalho M, Ford R, *et al*. Resilience is more than availability. In: Proc. of the 2011 New Security Paradigms Workshop. New York: ACM, 2011. 95−104. [doi: 10.1145/2073276.2073286]

[24] Diez O, Silva A. Resilience of cloud computing in critical systems. Quality and Reliability Engineering Int'l, 2014,30(3):397−412. [doi: 10.1002/qre.1579]

[25] Wolff E. Microservices: Flexible Software Architecture. Boston: Addison-Wesley Professional, 2016.

[26] Toffetti G, Brunner S, Blöchlinger M, *et al*. An architecture for self-managing microservices. In: Proc. of the 1st Int'l Workshop on Automated Incident Management in Cloud. New York: ACM, 2015. 19−24. [doi: 10.1145/2747470.2747474]

[27] Rusek M, Dwornicki G, Orłowski A. A decentralized system for load balancing of containerized microservices in the cloud. In: Proc. of the Int'l Conf. on Systems Science. Cham: Springer-Verlag, 2016. 142−152. [doi: 10.1007/978-3-319-48944-5_14]

[28] Soenen T, Tavernier W, Colle D, *et al*. Optimising microservice-based reliable NFV management & orchestration architectures. In: Proc. of the 2017 9th Int'l Workshop on Resilient Networks Design and Modeling (RNDM). Piscataway: IEEE, 2017. 1−7. [doi: 10.1109/RNDM.2017.8093034]

[29] Haselböck S, Weinreich R, Buchgeher G. Decision guidance models for microservices: Service discovery and fault tolerance. In: Proc. of the 5th European Conf. on the Engineering of Computer-based Systems. New York: ACM, 2017. 1−10. [doi: 10.1145/3123779.3123804]

[30] Heorhiadi V, Rajagopalan S, Jamjoom H, *et al*. Gremlin: Systematic resilience testing of microservices. In: Proc. of the 2016 IEEE 36th Int'l Conf. on Distributed Computing Systems (ICDCS). Piscataway: IEEE, 2016. 57−66. [doi: 10.1109/ICDCS.2016.11]

[31] Düllmann TF, van Hoorn A. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In: Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering Companion. New York: ACM, 2017. 171−172. [doi: 10.1145/3053600.3053627]

[32] Giedrimas V, Omanovic S, Alic D. The aspect of resilience in microservices-based software design. In: Proc. of the Federation of Int'l Conf. on Software Technologies: Applications and Foundations. Cham: Springer-Verlag, 2018. 589−595. [doi: 10.1007/978-3-030-04771-9_44]

[33] Van Hoorn A, Aleti A, Düllmann TF, *et al*. ORCAS: Efficient resilience benchmarking of microservice architectures. In: Proc. of the 2018 IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). Piscataway: IEEE, 2018. 146−147. [doi: 10.1109/ISSREW.2018.00-10]

[34] Jagiełło M, Rusek M, Karwowski W. Performance and resilience to failures of an cloud-based application: Monolithic and microservices-based architectures compared. In: Proc. of the IFIP Int'l Conf. on Computer Information Systems and Industrial Management. Cham: Springer-Verlag, 2019. 445−456. [doi: 10.1007/978-3-030-28957-7_37]

[35] Williams RC, Pandelios GJ, Behrens SG. Software risk evaluation (SRE) method description: Version 2.0. Pittsburgh: Software Engineering Institute, Carnegie Mellon University, 1999.

[36] Lee WS, Grosh DL, Tillman FA, *et al*. Fault tree analysis, methods, and applications—A review. IEEE Trans. on Reliability, 1985, 34(3):194−203. [doi: 10.1109/TR.1985.5222114]

[37] Alexander I. Misuse cases: Use cases with hostile intent. IEEE Software, 2003,20(1):58−66. [doi: 10.1109/MS.2003.1159030]

[38] Shostack A. Threat Modeling: Designing for Security. John Wiley & Sons, 2014.

[39] Stamatis DH. Failure Mode and Effect Analysis: FMEA from Theory to Execution. 2nd ed., Milwaukee: ASQ Quality Press, 2003.

[40] Lindvall M, Diep M, Klein M, *et al*. Safety-focused security requirements elicitation for medical device software. In: Proc. of the 2017 IEEE 25th Int'l Requirements Engineering Conf. (RE). Piscataway: IEEE, 2017. 134−143. [doi: 10.1109/RE.2017.21]

[41] Friedberg I, McLaughlin K, Smith P, *et al*. STPA-SafeSec: Safety and security analysis for cyber-physical systems. Journal of Information Security and Applications, 2017,34(2):183−196. [doi: 10.1016/j.jisa.2016.05.008]

[42] Basiri A, Behnam N, de Rooij R, *et al*. Chaos engineering. IEEE Software, 2016,33(3):35−41. [doi: 10.1109/MS.2016.60]

[43] Tucker H, Hochstein L, Jones N, *et al*. The business case for chaos engineering. IEEE Cloud Computing, 2018,5(3):45−54. [doi: 10.1109/MCC.2018.032591616]

[44] Blohowiak A, Basiri A, Hochstein L, *et al*. A platform for automating chaos experiments. In: Proc. of the 2016 IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). Piscataway: IEEE, 2016. 5−8. [doi: 10.1109/ISSREW.2016.52]

[45] Basiri A, Hochstein L, Jones N, *et al*. Automating chaos experiments in production. In: Proc. of the 2019 IEEE/ACM 41st Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Piscataway: IEEE, 2019. 31−40. [doi: 10.1109/ICSE-SEIP.2019.00012]

[46] Zhang L, Morin B, Haller P, *et al*. A chaos engineering system for live analysis and falsification of exception-handling in the JVM. IEEE Trans. on Software Engineering, 2019, PrePrints: 1−1. [doi: 10.1109/TSE.2019.2954871]

[47] Simonsson J, Zhang L, Morin B, *et al*. Observability and chaos engineering on system calls for containerized applications in docker. arXiv preprint arXiv:1907.13039, 2019.

[48] Salinas E. Tammy Bütow on chaos engineering. IEEE Software, 2018,35(5):125−128. [doi: 10.1109/MS.2018.3571246]

[49] ThoughtWorks. Technology radar vol.18. 2018. https://thoughtvorks.com/radar

[50] ThoughtWorks. Technology radar vol.20. 2019. https://thoughtvorks.com/radar

[51] Sharma B, Jayachandran P, Verma A, *et al*. CloudPD: Problem determination and diagnosis in shared dynamic clouds. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems & Networks. Piscataway: IEEE. 2013. 1−12. [doi: 10.1109/DSN.2013. 6575298]

[52] Bodik P, Goldszmidt M, Fox A, *et al*. Fingerprinting the datacenter: automated classification of performance crises. In: Proc. of the 5th European Conf. on Computer Systems. New York: ACM, 2010. 111−124. [doi: 10.1145/1755913.1755926]

[53] Cherkasova L, Kivanc O, Mi NF, *et al*. Automated anomaly detection and performance modeling of enterprise applications. ACM Trans. on Computer Systems, 2009,27(3):1−32. [doi: 10.1145/1629087.1629089]

[54] Duan S, Babu S, Munagala K. Fa: A system for automating failure diagnosis. In: Proc. of the 2009 IEEE 25th Int'l Conf. on Data Engineering. Piscataway: IEEE, 2009. 1012−1023. [doi: 10.1109/ICDE.2009.115]

[55] Kandula S, Mahajan R, Verkaik P, *et al*. Detailed diagnosis in enterprise networks. In: Proc. of the ACM SIGCOMM 2009 Conf. on Data communication. New York: ACM, 2009. 243−254. [doi: 10.1145/1592568.1592597]

[56] Nguyen H, Shen Z, Tan Y, *et al*. FChain: Toward black-box online fault localization for cloud systems. In: Proc. of the 2013 IEEE 33rd Int'l Conf. on Distributed Computing Systems. Piscataway: IEEE, 2013. 21−30. [doi: 10.1109/ICDCS.2013.26]

[57] Kandula S, Chandra R, Katabi D. What's going on? Learning communication rules in edge networks. ACM SIGCOMM Computer Communication Review, 2008,38(4):87−98. [doi: 10.1145/1402958.1402970]

[58] Nguyen H, Tan Y, Gu X. Pal: Propagation-aware anomaly localization for cloud hosted distributed applications. In: Proc. of the Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML 2011). New York: ACM, 2011. 1−8. [doi: 10.1145/2038633.2038634]

[59] Fonseca R, Porter G, Katz RH, *et al*. X-trace: A pervasive network tracing framework. In: Proc. of the 4th USENIX Symp. on Networked Systems Design & Implementation (NSDI 2007). USENIX, 2007. 271−284.

[60] Chen MY, Kiciman E, Fratkin E, *et al*. Pinpoint: Problem determination in large, dynamic Internet services. In: Proc. of the Int'l Conf. on Dependable Systems and Networks. Piscataway: IEEE, 2002. 595−604. [doi: 10.1109/DSN.2002.1029005]

[61] Zhao X, Zhang Y, Lion D, *et al*. Lprof: A non-intrusive request flow profiler for distributed systems. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2014). USENIX, 2014. 629−644.

[62] Chow M, Meisner D, Flinn J, *et al*. The mystery machine: End-to-end performance analysis of large-scale Internet services. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2014). USENIX, 2014. 217−231.

[63] Wang P, *et al*. CloudRanger: Root cause identification for cloud native systems. In: Proc. of the 2018 18th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGRID). Piscataway: IEEE, 2018. 492−502. [doi: 10.1109/CCGRID.2018.00076]

[64] Lin JJ, Chen P, Zheng Z. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Proc. of the Int'l Conf. on Service-oriented Computing. Cham: Springer-Verlag, 2018. 3−20. [doi: 10.1007/978-3-030-03596-9_1]

[65] Chen P, Qi Y, Hou D. CauseInfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. IEEE Trans. on Service Computing, 2019,12(2):214−230. [doi: 10.1109/TSC.2016.2607739]

[66] Standard Performance Evaluation Corporation. SPEC Benchmark. 2000. https://www.spec.org/benchmarks.html

[67] Transaction processing performance council. TPC Benchmark™C—Standard Specification Revision 5.11. 2010. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[68] Transaction Processing Performance Council. TPC Benchmark™W—Standard Specification Revision 2.0r. 2003. http://tpc.org/tpc_documents_current_versions/pdf/tpcw_v2.0.0.pdf

[69] European Telecommunications Standards Institute. ETSI GS NFV-TST 001 Network Functions Virtualisation (NFV); Pre-deployment Testing; Report on Validation of NFV Environments and Services. 2016. https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/001/01.01.01_60/gs_NFV-TST001v010101p.pdf

[70] Al-Masri E, Mahmoud QH. Qos-based discovery and ranking of Web services. In: Proc. of the 2007 16th Int'l Conf. on Computer Communications and Networks. Piscataway: IEEE, 2007. 529−534. [doi: 10.1109/ICCCN.2007.4317873]

[71] Zhang Y, Zheng Z, Lyu MR. Wsexpress: A QoS-aware search engine for Web services. In: Proc. of the 2010 IEEE Int'l Conf. on Web Services. Piscataway: IEEE, 2010. 91−98. [doi: 10.1109/ICWS.2010.20]

[72] Kalepu S, Krishnaswamy S, Loke SW. Verity: A QoS metric for selecting Web services and providers. In: Proc. of the 4th Int'l Conf. on Web Information Systems Engineering Workshops. Piscataway: IEEE, 2003. 131−139. [doi: 10.1109/WISEW.2003.1286795]

[73] Spirtes P, Clark G, Richard S. Causation, Prediction, and Search. 2nd ed., Cambridge: MIT Press, 1996. [doi: 10.1007/978-1-4612-2748-9]

[74] Pearl J. Causality: Models, Reasoning, and Inference. 2nd ed., New York: Cambridge University Press, 2009.

[75] Anderson TW, Amemiya Y. The asymptotic normal distribution of estimators in factor analysis under general conditions. The Annals of Statistics, 1988,16(2):759−771. [doi: 10.1214/aos/1176350834]

[76] Luo C, Lou JG, Lin Q, *et al*. Correlating events with time series for incident diagnosis. In: Proc. of the 20th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. New York: ACM, 2014. 1583−1592. [doi: 10.1145/2623330.2623374]

[77] Aderaldo CM, Mendonça NC, Pahl C, *et al*. Benchmark requirements for microservices architecture research. In: Proc. of the 1st Int'l Workshop on Establishing the Community-wide Infrastructure for Architecture-based Software Engineering. Piscataway: IEEE, 2017. 8−13. [doi: 10.1109/ECASE.2017.4]

[78] European telecommunications standards institute. ETSI GS NFV-REL 001, Network Functions Virtualisation (NFV): Resiliency Requirements. 2015. https://www.etsi.org/deliver/etsi_gs/NFV-REL/001_099/001/01.01.01_60/gs_NFV-REL001v010101p.pdf

[79] Thalheim J, Rodrigues A, Akkus IE, *et al*. Sieve: Actionable insights from monitored metrics in distributed systems. In: Proc. of the 18th ACM/IFIP/USENIX Middleware Conf. New York: ACM, 2017. 14−27. [doi: 10.1145/3135974.3135977]

殷康璘(1992－),男,博士,CCF 学生会员,主要研究领域为软件工程,智能运维.

杜庆峰(1968－),男,博士,教授,博士生导师,主要研究领域为软件工程与质量控制,机器学习与智能运维.