软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(7):3115−3140 [doi: 10.13328/j.cnki.jos.007104]

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

AmazeMap: 基于多层次影响图的微服务故障定位方法*

李亚晓, 李青山, 王 璐, 姜宇轩

(西安电子科技大学 计算机科学与技术学院, 陕西 西安 710071)

通信作者: 王璐, E-mail: wanglu@xidian.edu.cn

摘 要: 微服务软件系统由于其具有大量复杂的服务依赖关系和组件化模块, 一个服务发生故障往往造成与之相关的 1 个或多个服务发生故障, 导致故障定位的难度不断提高. 因此, 如何有效地检测系统故障、快速而准确地定位故障根因问题, 是当前微服务领域研究的重点. 现有研究一般通过分析故障对服务、指标的作用关系来构建故障关系模型, 但存在运维数据利用不充分、故障信息建模不全面、根因定位粒度粗等问题, 因此提出了 AmazeMap 方法. 该方法设计了多层次故障影响图建模方法以及基于多层次故障影响图的微服务故障定位方法. 其中: 多层次故障影响图建模方法通过挖掘系统运行时指标时序数据与链路数据, 考虑不同层次间的相互关系, 能够较全面地建模故障信息; 基于多层次故障影响图的微服务故障定位方法通过缩小故障影响范围, 从服务实例和指标两个方面发现根因, 输出最有可能的故障根因节点和指标序列. 基于开源基准微服务系统和 AIOps 挑战赛数据集, 从有效性和效率两个方面设计了微服务软件故障定位实验, 并与现有方法进行对比, 实验结果验证了 AmazeMap 的有效性、准确性和效率.

关键词: 微服务; 故障定位; 多层次故障影响图

中图法分类号: TP311

中文引用格式: 李亚晓, 李青山, 王璐, 姜宇轩. AmazeMap: 基于多层次影响图的微服务故障定位方法. 软件学报, 2024, 35(7): 3115–3140. http://www.jos.org.cn/1000-9825/7104.htm

英文引用格式: Li YX, Li QS, Wang L, Jiang YX. AmazeMap: Microservices Fault Localization Method Based on Multi-level Impact Graph. Ruan Jian Xue Bao/Journal of Software, 2024, 35(7): 3115–3140 (in Chinese). http://www.jos.org.cn/1000-9825/7104.htm

AmazeMap: Microservices Fault Localization Method Based on Multi-level Impact Graph

LI Ya-Xiao, LI Qing-Shan, WANG Lu, JIANG Yu-Xuan

(School of Computer Science and Technology, Xidian University, Xi'an 710071, China)

Abstract: Due to the large number of complex service dependencies and componentized modules, a failure in one service often causes one or more related services to fail, making it increasingly difficult to locate the cause of the failure. Therefore, how to effectively detect system faults and locate the root cause of faults quickly and accurately is the focus of current research in the field of microservices. Existing research generally builds a failure relationship model by analyzing the relationship between failures and services and metrics, but there are problems such as insufficient utilization of operation and maintenance data, incomplete modeling of fault information, coarse granularity of root cause localization, etc. Therefore, this study proposes AmazeMap, for which a multi-level fault impact graph modeling method and a microservice fault localization method are designed based on the fault impact graph. Specifically, the multi-level fault impact graph modeling method can comprehensively model the fault information by mining the collected temporal metric data and trace data while system running and considering the interrelationships between different levels; the fault localization method narrows the scope of fault impact, discovers the root cause from service instances and metrics, and finally outputs the most probable root cause of fault and

* 基金项目: 国家自然科学基金(62372351, U21B2015); 陕西省科协青年人才托举计划(20220113)
本文由"面向复杂软件的缺陷检测与修复技术"专题特约编辑张路教授、刘辉教授、姜佳君副研究员、王博博士推荐.
收稿时间: 2023-09-08; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-13

metrics sequence. Based on an open-source benchmark microservice system and the AIOps contest dataset, this study designs experiments to validate AmazeMap, and also compares it with the existing methods. The results confirm AmazeMap's effectiveness, accuracy, and efficiency.

Key words: microservice; fault localization; multi-level fault impact graph

微服务架构因增强了软件的敏捷性、弹性和可维护性, 已被广泛应用于我国航空航天领域(如航空工业集团制造云)、国防领域(如中电集团 SaaS 云平台)、电力领域(如国家电网 PKI 证书平台)和通信领域(如中国移动磐基 PaaS 平台)等重要支柱行业中. 如果不能准确、快速地对微服务故障进行定位、发现系统缺陷, 就会导致微服务系统运行状态持续恶化, 引发告警风暴, 造成严重后果. 例如: 2021 年, Facebook 出现大规模宕机, 全球无法使用近 7 h, 影响了超 8 000 万用户. 微服务架构具有高内聚、低耦合和去中心化的特点, 将系统拆分为一个个微服务, 而每个微服务均可进行多实例动态部署, 产生了大量不同状态的服务实例, 服务实例间异步通信产生的海量运维数据对系统分析提出挑战. 例如: 字节跳动公司的在线微服务数量超过 10 万,服务实例部署数量超过 1 000 万, 在巨大规模的压力下, 仅凭人工已无法从大量运维数据中快速发现和诊断异常情况[1].

微服务软件故障定位的研究工作大体分为故障检测和根因定位, 其中: 故障检测依赖于识别微服务软件运行过程中的异常[2]; 根因定位则是在故障检测结果的基础上, 通过相关算法展开进一步分析来确定故障发生的根本原因[3]. 而故障往往导致用户最终体验差, 即为软件缺陷. 目前, 微服务软件运行环境复杂多变、未知变化频发, 对建立一种能够保障微服务系统稳定运行的方法带来了以下两个难点.

(1) 多源异构运维数据难以时空融合. 指标数据通常是时间序列的形式, 链路数据通常是以服务间的调用关系的形式呈现, 反映微服务架构软件的空间信息, 且多源异构数据会随着时间的变化而动态变化. 同时, 多源异构数据之间是相互关联的. 因此, 从时间和空间角度融合这两类数据是必要的, 同时也是亟需解决的难点之一.

(2) 微服务架构软件故障难以细粒度定位. 大型微服务架构软件多实例容器化部署和独特的通信机制, 导致了监控数据种类和数据量的不断增加, 极大地影响了微服务故障检测与定位的精度, 定位粒度停留在服务级别. 然而, 该定位粒度并不能高效地帮助运维人员处理故障, 因此, 如何更细粒度定位故障, 是运维工作面临的难题.

目前, 研究工作针对上述两类难点问题展开研究, 但仍存在改进空间. 现有方法主要针对一类运维数据获取故障信息, 未考虑运维数据时空融合蕴含的故障信息, 导致故障的查全率低, 且检测与定位准确度较低. 例如, Nandi 团队提出的 OASIS 方法[4]只关注于日志数据, 忽略系统链路信息和度量指标的影响, 且其未考虑数据时间特性. 此外, 现有研究在进行故障定位时大多采用基于图的定位方法, 主要包括调用图[5]、因果图[6]和影响图[7,8]. 调用图和因果图针对服务调用关系和故障因果关系对系统建模, 不能全面且准确地定位根因. 而影响图能有效结合调用图和因果图的特点, 从故障表征、故障影响因素和故障根因的关系出发, 能够较为全面地描述故障相关信息, 提高定位的准确性和全面性.

基于此, 针对微服务软件故障定位面临的运维数据利用不充分、故障信息建模不全面、根因定位粒度粗等真实问题, 本文研究了基于多层次影响图的微服务故障定位方法, 提出了 AmazeMap. 该方法通过研究指标时序信息、指标因果信息和链路调用信息, 从时间和空间维度建模故障信息, 充分挖掘了运维数据的时间和空间特点, 实现多层次故障影响图构建, 并据此实现细粒度定位故障, 以达到高效定位故障并保证系统平稳运行的目的. 本文的主要贡献总结如下.

(1) 基于运维数据的多层次故障影响图建模方法. 现有的研究工作主要关注于某一类运维数据与故障间的映射关系, 从而设计出故障关系模型, 未从时间和空间维度建模故障信息. 相比于现有方法, AmazeMap 提出了三层影响图模型, 针对海量运维数据挖掘服务调用层、指标因果层和指标时序层之间的联系, 从时空角度全面建模故障信息, 实现度量指标和链路信息的时空融合; 从时间和空间维度解决了运维数据单一导致查全率低的问题, 为微服务故障定位提供模型基础, 提高软件故障检

测和根因定位的准确性.

(2) 基于多层次故障影响图的故障定位方法. 现有方法忽略了细粒度指标参数间的复杂关系会引起系统状态变化, 且大多工作的定位粒度为服务级别. AmazeMap 提出了基于动态阈值的软件故障检测方法, 结合异常分数和动态阈值有效检测故障发生范围, 精化故障影响图; 在此基础上, 设计了基于多层次影响图的故障根因定位方法, 分别从服务实例节点间调用关系和时序度量指标间因果关系两个层次逐步筛选故障节点, 精准定位故障节点位置和故障指标根因, 实现更细粒度的故障根因定位.

(3) 基于实验结果向运维人员提供优化建议. 通过在典型案例系统和数据集上开展实验, 通过与基于因果关系模型和基于影响关系模型的基准算法进行对比实验, 本文验证了 AmazeMap 的有效性和效率. 本方法可提醒软件运维人员某些故障容易诱发微服务软件缺陷. 软件运维人员可采取相应措施降低这些关键故障的发生概率, 从而使得微服务软件能够可靠、平稳地运行.

本文第 1 节介绍相关定义, 并简述方法框架. 第 2 节阐述多层影响图建模方法, 详细分析层与层之间的关联关系. 第 3 节介绍基于多层影响图模型的故障检测和根因分析算法. 第 4 节为实验分析, 验证算法的有效性和效率. 第 5 节介绍相关工作. 第 6 节内容总结与研究展望.

1 方法概览

定义 1(软件故障). 指软件无法正常运行或者严重影响软件业务正常执行的事件[9]. 本文中主要讨论微服务软件故障, 其故障类型根据故障表征分为功能失效故障和性能缺失故障[10].

定义 2(软件监控). 指动态地获取相关软件运行指标, 作为软件运维治理的重要组成部分, 主要用于反映软件运行状态. 而微服务软件监控包括了链路监控、度量指标监控和日志监控这 3 个部分.

定义 3(故障影响关系). 指在软件故障发生过程中, 不同服务实例和节点的指标间存在关联关系, 包括指标因果关系、指标时序关系和故障传播关系等.

定义 4(微服务软件故障类型). 根据软件故障的定义, 本文将微服务软件故障划分为功能失效故障和性能缺失故障, 其区别在于微服务软件系统的一部分业务功能是否失效. 在软件故障根源方面, 传统的单体软件故障和分布式软件故障根源分别集中在内部代码逻辑问题和分布式集群部署问题, 而微服务软件由于其去中心化的表现形式和轻量级的通信机制, 导致微服务软件故障根源的类型尤为复杂. 本文主要针对微服务软件特点, 并结合传统软件故障根源开展讨论, 将其分为内部缺陷故障、集群部署故障和服务依赖故障, 见表 1.

表 1 微服务软件故障类型划分表

类别	功能失效故障	性能缺失故障
内部缺陷	内存泄漏 堆栈溢出	I/O 操作时长过高 代码时间复杂度过高
集群部署	节点宕机 网络异常	负载过大 Docker 配置异常
服务依赖	版本不兼容 数据库连接阻塞	响应超时 丢包率过高

定义 5(基于调用图的故障根因定位方法). 该方法主要是通过采集软件运行时服务间相关调用的执行轨迹构建关系模型, 并利用轨迹差异对比[5]或树编辑距离等方法完成故障根因定位.

定义 6(基于因果图的故障定位方法). 通常利用度量指标与对应服务模型的相关性, 通过量化计算不同指标间的因果关系, 大多采用 PC 算法[11]完成因果图建模过程.

定义 7(基于影响图的故障定位方法[7,8]). 通过考虑故障传播、依赖关系和指标波动等对目标系统的影响, 分析量化不同故障表征、故障影响因素和故障根因间的关联关系, 进而构建故障影响图.

其次, 给出基于多层次影响图的微服务故障定位方法工作过程. 如图 1 所示, 其主要包含 3 部分工作: ① 运维数据采集; ② 多层次影响图模型构建; ③ 故障定位, 即故障检测及根因分析. 其中: 工作①为工作②

和③的基础，负责从系统运行状态中获取运维数据，提供数据基础；工作②为工作③的基础，也是本文核心之处，全面建模故障有关信息.

① 运维数据采集. 针对软件故障表征与多层影响图数据特点，利用软件监控手段，从微服务的度量指标数据(时间)和可执行链路数据(空间)两个方面对微服务进行全面故障数据采集，主要包括了对于服务调用信息、服务和节点部署信息、业务质量指标和性能度量指标等数据的获取；

② 多层次影响图模型构建. AmazeMap 创新性地从时空两个维度与指标因果、指标时序和故障传播关系这 3 个层次对故障影响图节点进行构建和关联. 一般的两变量格兰杰因果关系检验方法[12]无法有效地应用在复杂环境中存在的伪因果关系问题的两个以上相互作用的变量上，因此提出了基于多变量格兰杰因果关系检验方法对时序度量指标权重关系的量化计算方法，深入分析不同指标间的作用关系，最终输出一个有向带权图模型，提高后续软件故障检测和根因定位的准确性；

③ 故障定位(故障检测和根因分析). 基于上述多层次影响图模型，首先提出了基于动态阈值的软件故障检测方法，其次提出了基于影响图的故障根因定位方法，精确定位故障节点位置和故障指标根因.

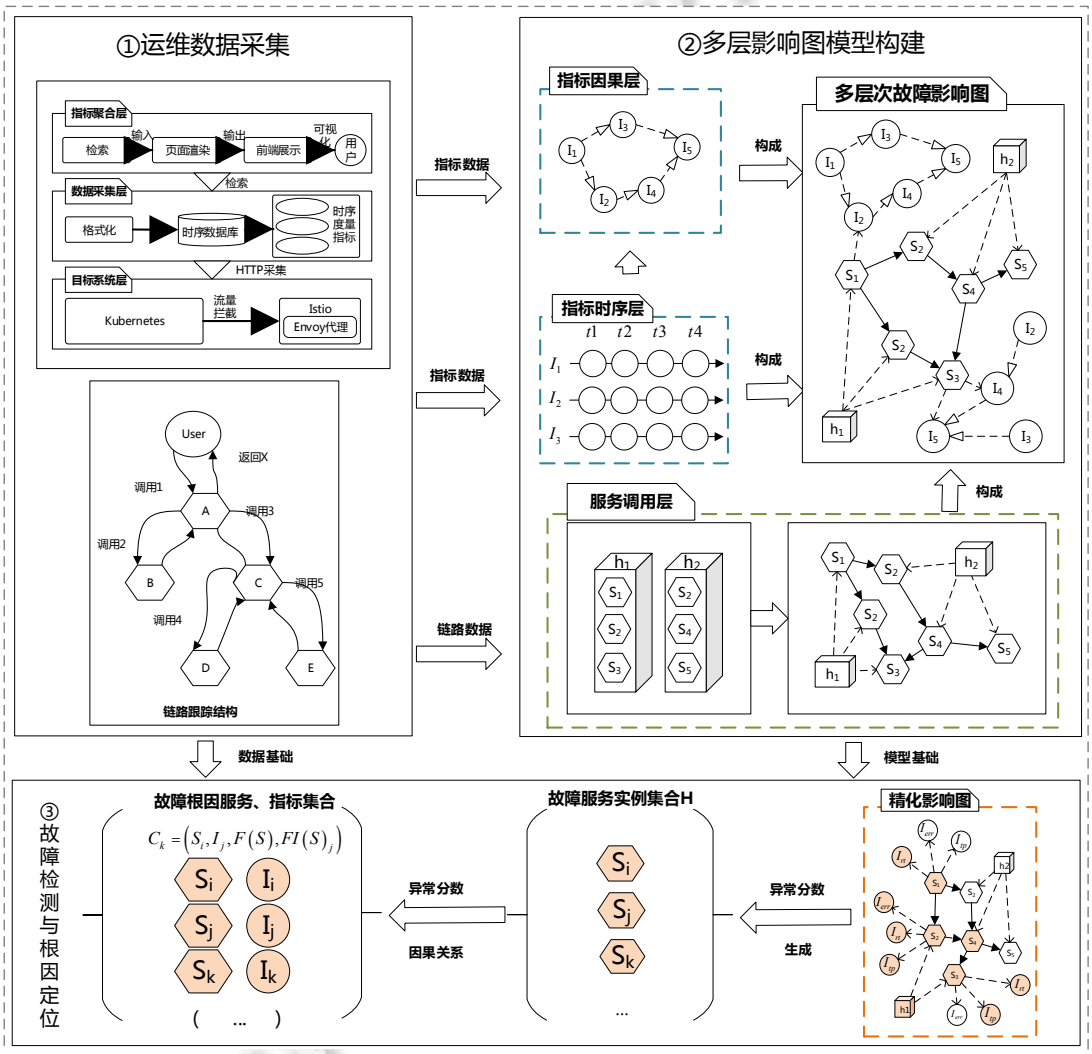


图 1 AmazeMap 方法概览

2 基于运维数据的多层次故障影响图建模方法

2.1 运维数据采集

为了全面准确地度量微服务软件运行状态, 本文从时间和空间角度对微服务系统进行全面故障数据采集, 包括度量指标数据和可执行链路数据两个方面. 由于微服务架构的去中心化和轻量级通信的特点及其在基础设施层和应用服务层与传统软件存在较大差异, 给微服务的链路数据采集过程造成极大的阻碍. 在基础设施层, 一个服务组件存在多个实例部署, 并且部署在不同的物理节点上, 难以有效检测在一条请求链路中是哪个节点的哪个服务实例存在故障. 在应用服务层, 每个微服务组件独立承担一部分业务功能, 且同一微服务的种类和服务实例繁多, 相互调用错综复杂, 很难有效获取清晰的请求调用链路. 本文结合了微服务软件在基础设施层和应用服务层的特点, 通过动态插桩方法采集服务部署和交互信息, 构建跨节点的可执行链路. 该方法能够将侵入代码限制在软件堆栈的一个足够低的级别, 保证了方法的应用程序透明性, 从而实现链路数据的全面部署和持续采集, 最终达到微服务可执行链路数据采集的目的.

该方法主要分为设计链路模型、刻画服务调用信息、采集链路数据并完成模型构建这 3 个部分.

- 首先, 通过动态插桩方法采集服务部署和交互信息, 构建跨节点的可执行链路, 以实现链路数据的采集. 本文通过对微服务软件系统内部相关服务的研究, 同时, 本文结合了 Dapper 跟踪树的理论和时间跨度, 将微服务间服务调用关系视作一个跟踪树的结构.
- 其次, 本文通过多元组和有向边分别表示服务组件实例和服务调用信息.
- 最后, 在设计链路模型和刻画服务调用信息的基础上, 利用动态插桩的思想, 通过 Kafka 和 HBase 等组件实现对服务调用信息的采集. 采用基于宽度优先搜索方法, 对父子调用关系进行分析, 最终完成了可执行链路的构建.

随着微服务架构的广泛应用, 多实例容器化部署和独特的通信机制导致了监控数据种类和数据量的不断增加, 同时, 相关指标的生命周期也逐渐缩短, 传统的软件监控工具已无法满足微服务软件度量指标采集的要求. 因此, 本文提出一种基于时序数据库的多维度度量指标采集方法, 在 USE 方法(utilization saturation and errors method)的理论原则下, 结合 Prometheus 以及 Kubernetes 容器实践, 提出了针对实际物理资源、服务部署和运行资源的度量指标监控方法. 该方法首先通过明确划分微服务软件系统的不同维度的指标, 然后利用 HTTP 协议持续性抓取被监控组件的状态, 不需要任何 SDK 或其他集成方式, 适用于微服务虚拟化环境的要求, 完成了对于微服务软件系统在基础设施层、服务应用层以及网络层等多个层次在一定时间周期内的业务质量指标和性能度量指标, 从而能够有效甄别资源瓶颈, 反映微服务软件系统下的故障问题.

该方法将时序度量指标监控过程分为目标系统层、数据采集层和指标聚合层这 3 个层次.

- 首先, 在目标系统层, 将目标微服务软件系统部署于 Kubernetes 平台上, 通过服务网格化的形式设置 Envoy 动态代理, 实现对微服务间的交互进行流量拦截, 并通过 Kubernetes 平台自带的监控组件实时获取系统基础设施(包括 CPU、内存、磁盘、网络和线程等相关维度)的监控指标.
- 然后, 在数据采集层, 构建基于时序数据库的数据采集器, 通过被监控系统暴露 HTTP 服务接口的方式, 周期性地抓取监控数据, 使得被监控系统完全独立于监控系统之外, 无需感知监控模块, 进一步地降低耦合; 在此基础上, 针对时序数据库特点, 完成了环境部署指标、中间关联指标以及用户感应指标的多元组表示进行持久化存储.
- 最后, 在指标聚合层, 本文利用 ECharts 组件对度量指标进行聚合统计, 有效反映微服务系统的运行状态, 提高软件运维效率.

2.2 多层次影响图建模方法

现有的研究工作主要关注于分析链路、服务性能指标变化与故障间的映射关系, 忽略了细粒度指标参数间的复杂相互作用会引起系统状态变化, 在一定程度上影响了故障表征, 从而导致故障映射关系存在误差, 影响后续故障定位的准确性. 因此, 本文设计了多层次故障影响图模型, 如图 2 所示, 从服务调用层、指标时

序层和指标因果层这 3 个层次对故障影响图节点进行构建和关联; 然后提出了基于多变量格兰杰因果关系检验方法, 对时序度量指标权重关系进行量化计算, 深入分析不同指标间的作用关系; 最终输出一个有向带权图模型, 为后续故障定位提供模型支撑.

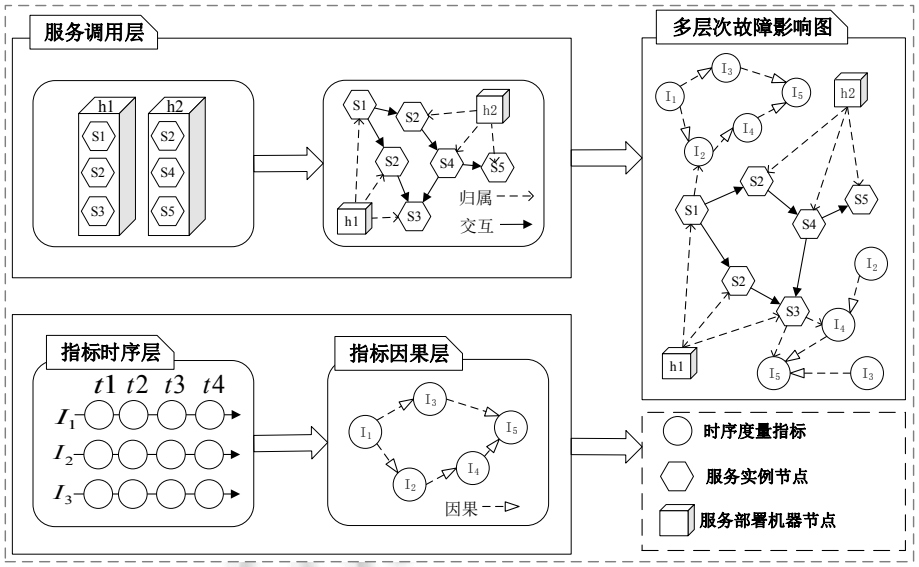


图 2 多层次故障影响图模型

2.2.1 多层次影响图模型

(1) 服务调用层

AmazeMap 通过分析特定时刻下的可执行链路和服务部署状态, 挖掘链路数据空间信息, 以服务实例作为故障影响图的图节点. 不同服务实例之间用服务调用关系和部署关系作为连接边, 而每个服务实例对应于不同类型的服务组件和部署机器节点. 同机器节点下的不同服务实例节点存在多个相同数据的度量指标, 即 CPU 使用率、内存占用率和网络带宽等与机器节点相关的度量指标. 服务调用层能够有效解决在同一条可执行链路、不同可执行链路、同服务部署机器节点下以及跨节点状态下的软件故障传播问题和故障事件覆盖率不足的问题. 例如, 实例 $S1$ 和实例 $S2$ 同时部署于 $h1$ 节点, 实例 $S1$ 归属于服务调用链路 1, 实例 $S2$ 归属于链路 2, 当实例 $S1$ 存在资源受限异常, 链路 1 会发生异常事件, 同时有可能导致 $h1$ 故障或宕机, 从而链路 2 也会由于实例 $S2$ 的运行异常而产生异常事件.

(2) 指标时序层

由于软件故障的传播并非是一个瞬时的事件, 单纯某一时刻的服务调用信息和度量指标数据无法精确建模故障的传播和影响过程. 因此, AmazeMap 主要考虑软件故障时序关系和传播延迟的影响, 结合指标的历史数据和实时数据, 确定不同指标间的时序关系. 该时间序列为最大的度量指标影响时间范围, 确保涵盖了所有因度量指标变化而引起的其他指标波动. 然后, 通过对比分析不同时刻下指标的波动情况, 进而量化计算出指标间因果关系, 挖掘出在一段时间内不同度量指标间的故障传播方向, 从而解决了由于微服务轻量级通信导致的故障传播时延问题; 同时, 能够细粒度地判断微服务软件系统由于软件故障所引发的内部指标变化趋势, 为后续故障检测和细粒度根因定位提供数据支撑.

(3) 指标因果层

本文通过分析时序度量指标间的因果关系, 在指标时序关系的基础上, 将每个图节点表示为某种度量指标在一段时间内的持续监控数据; 同时, 将度量指标间的因果关系作为故障影响图中每个度量指标节点的带权有向边, 用于表示某种指标受另一种指标的影响程度, 从而能够深入了解软件运行过程中不同度量指标变化的关联性. 由于用户感应指标能够切实反映软件系统业务的执行状态, 并且一个用户感应指标的异常通常

是由若干环境部署指标和中间关联指标共同作用的最终结果, 因此, AmazeMap 将用户感应指标作为微服务软件系统中不同度量指标相互作用导致的最终结果, 即故障表征. 例如: 用户每秒请求量的增加导致了 CPU 使用率、CPU 负载、内存占用率和数据库操作数等指标的变化; 同时, CPU 负载在一定程度上也影响了 CPU 使用率, 数据库操作数的增加也导致了线程数的增加, 最终导致了服务响应时间的延长, 引发了微服务软件故障.

2.2.2 节点构建与关联

在节点构建方面, 首先, 将不同时间段微服务软件系统的可执行链路和时序度量指标数据集表示为 I_i^t, t=0,1,…,T, i=1,2,…,N, 其中, T 为软件故障数据种类, t 表示不同时刻下监控数据的时间序列. 然后, 结合时序度量指标、服务调用和基础设施部署状态生成一个完全有向图, 用 $G(V,E,I)$表示某一时刻的微服务软件运行状态. 其中: V 表示图节点, 用 V_i, i=1,…,N 表示, 包括服务实例节点和时序度量指标节点; E 是故障影响图的边集合, 表示服务调用关系和指标因果关系; I 则代表多维度的度量指标集合, 包括用户感应指标、中间关联指标和环境部署指标这 3 类.

针对服务实例节点, 主要以可执行链路信息和服务部署信息作为服务实例节点的关键数据, 每一个服务实例节点都包含了少量的额外信息, 例如实例部署节点、服务实例种类等, 用于准确定位该服务实例的位置. 此外, 不同服务实例节点间的边用服务调用关系和服务同一机器节点部署关系表示. 根据上述定义, 能够输出一个反映服务调用信息的有向图 G_T.

时序度量指标节点代表一段时间内持续的指标监控数据, 主要以度量指标的实时数据和一段时间内的指标波动情况作为关键数据. 本文通过设置滑动窗口获取度量指标数据样本. 每个时序度量指标节点表示为 $I_i = I_i^{t-\tau},...,I_i^t$, 其中包括了当前时刻的瞬时数据和在时间窗口 τ内的监控数据, 即故障影响的最大传播时间, 保证了故障传播局限在这一段时间内. 同时, 标注出由于软件故障发生造成的对应度量指标波动过大的时间节点和数据, 根据不同度量指标的变化顺序反映指标时序关系, 表示为 $TR(I_i^{t1},I_j^{t2})$. 此外, 不同度量指标节点间的边表示时序和因果关系, 并通过量化因果关系的手段计算不同度量指标节点间的影响关系, 以此作为有向边的带权值 $P(S_i,I_i^{t1},I_j^{t2})$. 如果 $P(S_i,I_i^{t1},I_j^{t2})$ 在时序关系上无法满足 $t1<t2$, 或者在因果关系上无法满足 $P(S_i,I_i^{t1},I_j^{t2})>0$ 的条件, 代表 I_j^{t1} 对 I_j^{t2} 间不存在影响关系, 则从给定完全有向图中去除从 I_i 指向 I_j 的有向边. 不断循环该过程, 最终输出一个反映不同指标间时序因果关系的带权有向图 G_m, 并作为故障影响图的重要组成部分, 具体的度量指标的时序因果图 G_m 的建模算法 1 所示.

算法 1. 时序度量指标的因果关系图建模方法.

输入: 时序度量指标数据集 *metricSet*, 时序时间窗口大小为 *timeWindow*.

输出: 时序度量指标因果关系图(带权有向图).

```
1.  S=constraintsByTime(metricSet,timeWindow)          //基于时间窗口获取数据集
2.  M=constraintsByFaultOrChangeEvent(S)               //初步筛选故障影响的指标集合
3.  len=S.size(·)                                      //得到数据集中指标种类数
4.  I=init(S)                                          //得到所有时序度量指标值
5.  G_m=initGraph(M,len)                               //构建完全有向图 G_m
6.  for i←1 to len do
7.      for j←1 to len do
8.          p=calculateCausalByGC(I_i,I_j)             //通过 GC 算法计算因果关系
9.          if p=0 then
10.             deleteGraphEdge(G_m,i,j)               //删除从 I_i 到 I_j 的边
11.         else
12.             if I_i∈M && I_j∈M then                 //判断 I_i 和 I_j 是否故障指标
```

```
13.                t1=getFaultOrChangeTime(F,i)          //获取 I_i 发生故障的时间
14.                t2=getFaultOrChangeTime(F,j)          //获取 I_j 发生故障的时间
15.                if t1<t2 then
16.                    value=calPearsonValue(I_i,I_j)    //量化因果关系
17.                    insertValueofEdge(G_m,i,j,value)  //将因果值赋给 I_i 到 I_j 的边
18.                else
19.                    deleteGraphEdge(G_m,i,j)          //删除从 I_i 到 I_j 的边
20.            endif
21.        endif
22.    endfor
23. endfor
```

在完成多层次故障影响图模型节点的构建和关联之后, 本文提出了时序度量指标因果关系量化计算方法. 本文采用基于多变量格兰杰因果关系检验方法判断度量指标节点间的因果关系作为影响关系; 然后, 通过皮尔逊相关系数对影响图进行加权处理, 以此量化软件故障在指标节点上的影响关系, 即故障影响图中不同指标节点间的带权有向边的权重值.

一般的两变量格兰杰因果关系检验方法仅适用于二元时间序列, 无法有效地应用在存在两个以上相互作用的变量复杂环境中存在的伪因果关系问题, 主要分为两种情况.

- 其一, 在 3 个变量 X, Y 和 Z 同时存在的前提下, 变量 X 是变量 Y 的原因变量, 变量 Z 又是变量 X 的原因变量. 然而传统的格兰杰因果关系检验没有考虑第 3 种变量 Z 在相互作用过程中的影响, 无法判断变量 X 与变量 Y 之间存在的是直接因果关系或间接因果关系, 从而引发检验结果错误, 导致 3 个变量之间的因果关系混淆.
- 其二, 当变量 X 是变量 Y 和 Z 的原因变量, 可知变量 Y 与变量 Z 同源. 当计算变量 Y 与变量 Z 之间的格兰杰因果关系的过程中, 由于变量 Y 与变量 Z 的数据变化都来自同一个原因变量, 错误判定变量 Y 与 Z 间存在因果关系.

上述两种情况都违背了基础假设, 即不同原因变量间存在相互关联, 从而引发检验结果出错. 此外, 在现实场景下, 也无法控制所有与该相互作用过程相关的所有剩余变量, 极大地影响了格兰杰因果关系检验方法的准确性. 因此, 为了完成多层次故障影响图建模, 需要针对复杂大规模时序度量指标进行因果关系量化计算. 本文提出通过引入条件变量的方式开展多变量格兰杰因果关系检验, 在一定程度上解决了伪因果关系问题, 提高故障影响图边权重的准确性和有效性.

针对变量 X, Y 和 Z 的时间序列, 通过构建三变量时间序列的回归方程, 包括基准方程和比较方程两类, 分析出变量 Y 与变量 Z 间的格兰杰因果关系. 其中, 构建的基准方程如公式(1)所示, 构建的比较方程如公式(2)所示.

$$\begin{cases} X(t)=\sum_{i=1}^{p}\alpha_i x_{t-i}+\sum_{j=1}^{p}\beta_j y_{t-j}+\varepsilon_1(t) \\ Y(t)=\sum_{i=1}^{p}\chi_i x_{t-i}+\sum_{j=1}^{p}\delta_j y_{t-j}+\varepsilon_2(t) \end{cases} \tag{1}$$

$$\begin{cases} X(t)=\sum_{i=1}^{p}\alpha_i x_{t-i}+\sum_{j=1}^{p}\beta_j y_{t-j}+\sum_{k=1}^{p}\eta_k z_{t-k}+\varepsilon_3(t) \\ Y(t)=\sum_{i=1}^{p}\chi_i x_{t-i}+\sum_{j=1}^{p}\delta_j y_{t-j}+\sum_{k=1}^{p}\mu_k z_{t-k}+\varepsilon_4(t) \end{cases} \tag{2}$$

然后, 本文将变量 X 作为条件变量, 为分析变量 Z 是否为变量 Y 的原因变量, 构建因果统计量方程如公式(3)所示.

$$F_{z\to y|x}=\frac{(RSS_0-RSS_1)/p}{RSS_1/(N-2p-1)} \tag{3}$$

在此基础上, 针对传统格兰杰因果关系检验在多变量环境下产生的两类伪因果关系问题展开讨论. 当变量 X, Y 与 Z 的关系符合直接与间接因果关系时, 即只有变量 X 作为条件变量与变量 Y 与 Z 存在直接因果关系. 此时, z_{t-k} 中不包括额外的因果关系作用信息, 对 $Y(t)$没有影响. 而当变量 X, Y 与 Z 的关系符合同源关系时, 变量 Y 与变量 Z 同时依赖于原因变量 X. 此时, x_{t-i} 中已经包含了所有的因果作用关系信息, 而 z_{t-k} 没有提供额外的信息, 对 $Y(t)$没有影响. 综合以上两种情况可以得出, 公式(1)中的$\varepsilon_2(t)$应该等于公式(2)中的$\varepsilon_4(t)$, 进而导致 $F_{z\to y|x}$=0; 倘若变量 Z 与变量 Y 之间存在直接因果关系, 则表示 z_{t-k} 中包含了一部分变量 Y 因果作用信息, 使得 $\varepsilon_2(t)>\varepsilon_4(t)$, 从而导致 $F_{z\to y|x}>0$. 此外, 本文通过将复杂多变量的场景拆分为三变量的场景, 通过划分 3 个变量的结构逐步解决伪因果关系.

此外, 由于时序度量指标时间序列数据的波动性和随机性, 当变量之间不存在格兰杰因果关系状态时, 也会发生 $F_{z\to y|x}>0$ 的情况, 与原假设不符. 因此, 为了排除数据随机干扰的问题, 本文采用基于蒙特卡罗模拟的方法, 设置不同情况下因果统计量的置信阈值. 以公式(2)为例, 首先将已有的z_t时间序列随机打乱顺序, 输出得到新的时间序列 $\hat{z}_t$, 从而破坏了变量 Z 与变量 Y 之间原有的因果关系, 确保新的时间序列 z_t 对于变量 Y 无作用关系, 假定满足原假设 $F_{z\to y|x}$=0. 然后, 通过重复打乱时间序列 $\hat{z}_t$, 基于公式(2)计算因果统计量, 将计算输出结果进行升序排列. 最终, 根据统计学原理,选择满足 95% 的分位数作为该因果统计量的置信阈值. 最终, 针对存在因果关系的两个时序度量指标, 采用皮尔逊相关性系数量化计算其相关性, 以此作为因果关系边的权重值.

本文重点关注微服务软件故障的传播形式, 分别从跨指标传播和跨服务传播两个角度展开分析, 有效关联服务调用、服务部署、指标时序以及指标因果关系, 完成对不同层次的故障影响图模型间的转换. 在跨指标传播角度, 由于不同度量指标间存在冗余信息, 即双向关联关系, 单纯依赖指标数值波动难以有效判断不同指标间的因果关系和传播情况. 因此, 本文结合指标时序关系, 在因果关系量化计算的基础上, 通过不同指标发生异常波动的前后次序来判断不同指标间的指向关系, 从而保证两个度量指标间为单向带权边. 在跨服务传播角度, 根据服务实例在基础设施层和服务应用层所担任的角色, 本文将服务调用可执行链路和服务部署状态作为微服务软件故障跨服务传播的主要途径. 其中: 服务调用可执行链路主要利用用户感应指标反映不同服务实例节点的运行状态, 以此确定软件故障链路; 服务部署状态则通过环境部署指标的变化来影响同部署机器节点上不同服务实例间的故障传播状态.

3 基于多层次故障影响图的故障定位方法

3.1 基于动态阈值的软件故障检测方法

(1) 软件异常分数评估

本文从可执行链路耗时评估和服务指标评估两部分对微服务软件异常分数进行评估, 具体如图 3 所示. 首先, 通过对可执行链路耗时进行异常分析评估, 分析故障链路集合; 在此基础上, 根据该集合中存在的服务调用关系进行遍历, 初步筛选出故障服务集合; 最后, 对故障服务集合中的微服务的用户感应指标进行异常分数评估, 最终筛选出引起软件异常的关键服务实例, 从而缩小故障检测范围.

- 异常分数设计. 当微服务软件系统具体某一条可执行链路发生故障时, 势必会导致链路耗时的异常波动. 本文结合了自回归模型和加权移动平均模型的特点构建回归模型, 通过量化计算历史数据与当前数据间的关联关系, 并解决历史数据中不同时刻对于当前数据的影响程度不同的问题, 对链路耗时的当前值进行预测. 其中, 移动平均过程消除随机波动对预测值的影响, 从而确保该模型的预测效果更为准确和高效. 在此基础上, 获得该链路耗时的预测值. 将得到与实际链路耗时的差值与预测值比较得到预测偏差比. 预测偏差比值作为该链路的异常分数, 当该预测耗时与当前时刻的实际链路耗时有较大差距时, 即异常分数超过阈值时, 则判定为该链路发生了故障. 本文通过异常分数评估

方法评估可执行链路的异常程度，并将异常分数过大的链路和执行失败的链路加入故障链路集合 *TS* 中.

- 故障服务集合构建. 本文设置了故障服务集合 *FS*, 通过遍历故障链路的服务调用轨迹，并将经过的所有服务实例加入故障服务集合 *FS* 中；考虑到微服务软件故障在物理资源层面的故障传播问题，本文将与故障服务集合 *FS* 所有的服务实例部署在同一物理节点上的其他服务实例加入故障服务集合 *FS* 中，确保该集合中涵盖了某特点故障的故障表征和故障根因.
- 精化影响图. 本文利用异常分数评估方法依次对故障服务集合 *FS* 中的所有服务实例的用户感应指标进行异常评估，包括方法响应时间、吞吐量以及错误率这 3 类度量指标. 将异常分数未达到阈值的服务实例从故障服务集合 *FS* 中筛除，精化故障服务集合. 如图 4 所示，在第 3 节多层次故障影响图的基础上，根据故障服务集合筛选出软件故障影响范围，以此精化故障影响图，减少后续故障根因定位过程的工作量，加快故障根因定位速度.

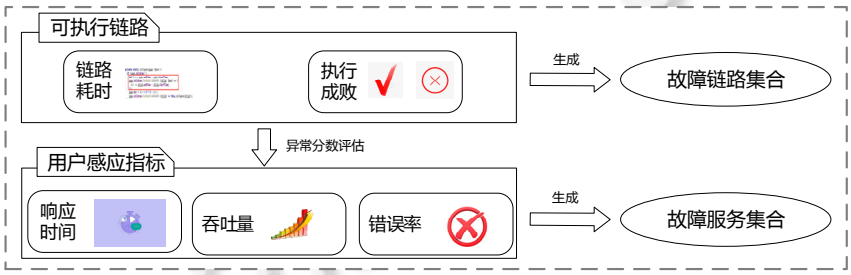


图 3 软件异常分数评估结构图

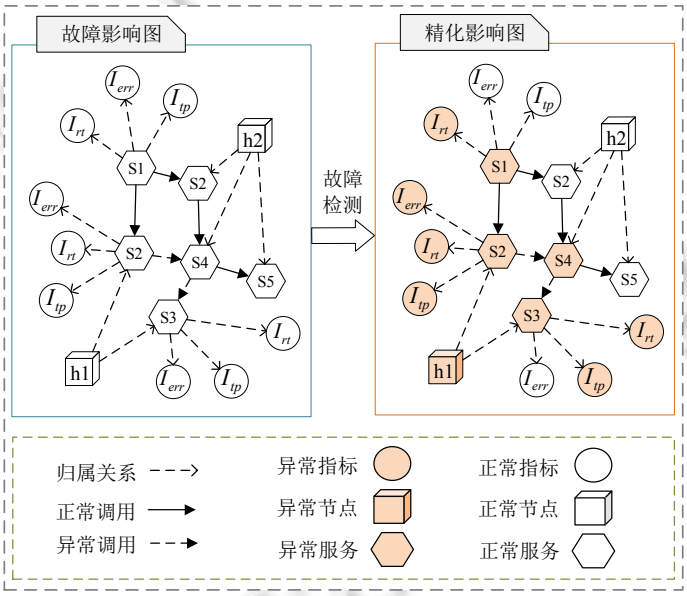


图 4 精化故障影响图

(2) 基于累积分布函数的动态阈值计算

本文结合了累积分布函数用于描述异常分数的分布情况，将离散数据进行平滑处理，输出连续曲线，并参考连续函数曲率的定义分析异常分数增长缓慢的位置作为动态阈值. 已知：异常分数阈值越高，被判定为故障服务实例节点的数量越少，有可能会导致故障节点作为正常节点被剔除出故障服务集合 *FS*, 从而影响故障定位的准确性. 因此，本文将故障服务集合 *FS* 中的服务实例在每个时刻的用户感知指标作为基于曲率的动

态阈值计算方法的输入. 首先, 基于对应指标的异常分数生成对应的离散数据, 得到不同指标的预测值和实际值; 然后, 根据加权移动对离散数据进行平滑处理, 得到定义平滑曲线 *Ds*; 最终生成一个以预测偏差值为 *X* 轴、被判定为正常指标节点占所有节点的比值为 *Y* 轴的 CDF 图.

$$Ds=\{(x_i,y_i)|x_i,y_i\geqslant 0\} \tag{4}$$

其次, 为了确保基于累积分布函数曲率的动态阈值计算方法的通用性, 不受用户感应指标变化的干扰, 如公式(5)所示, 对该平滑曲线 *Ds* 进行归一化处理.

$$\begin{cases} x_i=(x_i-x_{\min})/(x_{\max}-x_{\min}) \\ y_i=(y_i-y_{\min})/(y_{\max}-y_{\min}) \end{cases} \tag{5}$$

然后, 如公式(6)所示, 用 *y*–*x* 取代 *y* 值, 获得新的平滑曲线, 用 *Dd* 表示.

$$\begin{cases} Dd=\{(x_{di},y_{di})\in R^2\mid x_{di},y_{di}\geqslant 0\} \\ x_{di}=x_i \\ y_{di}=(y_i-x_i) \end{cases} \tag{6}$$

该方法能够有效去除预测偏差比与正常节点率按线性变化的冗余数据.

在此基础上, 为了找到最优阈值点, 本文在归一化后的连续函数上进行局部最大值计算, 如公式(7)所示, 生成以预测误差比为 *X* 轴、正常节点率为 *Y* 轴的连续平滑曲线 *Dm*, 将局部最大点作为最优阈值点的候补节点, 用于表示增长率开始下降的位置.

$$\begin{cases} Dm=\{(x_{mi},y_{mi})\in R^2\mid x_{mi},y_{mi}\geqslant 0\} \\ x_{mi}=x_{di} \\ y_{mi}=y_{di}\mid y_{di-1}<y_{di},y_{di+1}<y_{di} \end{cases} \tag{7}$$

此外, 在得到候补阈值点之后, 需要定义一个唯一阈值, 作为 *Dm* 连续值与灵敏参数 ϑ 间的平均差值, 具体计算过程如公式(8)所示.

$$T_{mi}=y_{mi}-\vartheta\cdot\frac{\sum_{i=1}^{n-1}(x_{i+1}-x_i)}{n-1} \tag{8}$$

灵敏参数 ϑ 用于调整计算动态阈值的速度. 当灵敏参数 ϑ 值越大时, 动态阈值的计算时间越慢; 反之, 则越快. 本文中, 该灵敏参数默认设置为 ϑ=1.

最后, 本文通过候选阈值与上述 T_{mi} 相对比, 筛选出符合条件的阈值. 当存在(x_{dj},y_{dj}), $j>i$, 且该值在 T_{mi} 之下时, 此刻对应的局部最大值为最优阈值 $x=x_{mi}$. 此外, 该动态阈值持续到下一个超过 T_{mi} 的局部最大值点.

3.2 基于影响图的故障根因定位方法

在上述精化后的多层次故障影响图的基础上, 本节设计了基于影响图的故障根因定位方法. 首先对该故障根因定位方法所涉及的问题进行阐述, 并完成符号定义(见表 2).

表 2 微服务故障根因定位符号定义表

符号	定义
S_i	第 i 个服务实例
$I(S_i)$	某个服务实例节点 S_i 的监控指标集合
$I(S_i)_j$	某个服务实例节点 S_i 的监控指标集合第 j 个指标
$F(S_i)$	第 i 个服务实例节点的异常分数
$FI(S_i)_j$	第 i 个服务实例节点的第 j 个用户感应指标的异常分数
$P(S_i,I_i,I_j)$	第 i 个服务实例节点上指标 j 和指标 k 间的因果关系
C_i	故障根因集合中第 i 个根因(以多元组表示)

(1) 基于调用关系的故障服务节点定位

为了从多层次故障影响图中筛选出可能是本次故障影响的根因故障节点, 本节提出了基于服务调用关系的故障节点定位方法, 用于计算不同服务实例节点的异常程度, 并筛选出可能导致软件故障的根因节点位置

集合. 本文采用了基于特征向量中心性的 PageRank 方法, 该方法考虑到了不同服务实例节点的重要程度对其他相邻节点的影响. 具体过程如下.

- 首先, 对服务实例节点的用户感应指标的异常分数进行评估, 包括方法响应时间、吞吐量以及错误率这 3 类度量指标, 并将异常度量指标的值归一化到[0,1]区间内. 该异常分数表示为对应指标当前时刻实际值与预测值间的残差值.
- 其次, 将 3 种用户感应指标归一化后的异常分数之和作为服务实例节点 S_i 的异常分数的初始值, 如公式(9)所示.

$$F(S_i)=\sum_{j=1}^{3}FI(S_i)_j \tag{9}$$

- 然后, 构建一个 $n\times1$ 的矩阵向量 v, 用于表示每个服务实例节点的初始异常分数, 如公式(10)所示.

$$v=[F(S_1),F(S_2),\ldots,F(S_n)]^T \tag{10}$$

此外, 在多层次故障影响图模型的基础上, 根据服务实例间的调用关系和同一机器节点部署关系构建一个 $n\times n$ 的关联矩阵 M, 其中, $M=(M_{ij})$. 在本文中, 采用皮尔逊相关性系数计算方法, 对存在影响关系的不同服务实例节点的响应时间序列作相关性计算, 并将量化后的每个服务实例节点的相关性占与该节点相关的所有节点的相关性之和的比值作为不同服务实例节点间的影响概率. 具体如公式(11)所示.

$$M_{ij}=\frac{P(S_i,S_j)}{\sum_{k\in M(i)}P(S_i,S_k)} \tag{11}$$

其中, M_{ij} 表示第 i 个服务实例对第 j 个服务实例节点的影响概率, $P(S_i,S_j)$表示服务实例节点 S_i 对 S_j 的影响关系量化值, $k\in M(i)$表示服务实例 S_i 会对 S_k 产生影响. 如果 M_{ij} 的值大于 0, 表示服务实例 S_i 与服务实例 S_j 之间存在影响关系; 如果值为 0, 则表示两个服务实例不存在相互作用关系.

然后, 将上述向量 v 和矩阵 M 带入迭代方程, 如公式(12)所示, 采用幂次迭代算法计算每个服务实例节点的异常分数, 直至该数值趋于稳定, 表示迭代结束.

$$v'=dMv+\frac{1-d}{N}U \tag{12}$$

其中, N 表示所有服务实例节点的数量; d 是阻尼系数, 在本方法中, 默认 d=0.85; 而$(1-d)/N$ 表示某一服务实例节点被随机访问的概率. 需要注意的是: 通过反复迭代趋于稳定状态需要满足约束条件$|v'-v|<\psi$, 表示迭代结束. 其中: v'表示第 k 次迭代后计算得到的异常分数; v 表示前一次的异常分数; ψ为迭代停止, 异常分数趋于稳定的阈值, 在本文中, 默认ψ=0.01.

最后, 通过对每个服务实例节点的稳定异常分数排序, 并筛选出异常分数最高的前 10 个故障实例节点, 加入故障服务实例节点集合 H 中. 最终, 基于用户感应指标的服务实例节点的异常分数进行排序, 筛选出前 10 个异常分数最高的异常服务实例节点集合作为微服务软件系统的故障根因节点集合.

(2) 基于因果关系的故障指标根因定位

由于不同的度量指标间存在复杂的因果关系, 度量指标之间会相互影响, 从而导致更大的软件故障. 为了提高软件运维人员的故障排查效率, 本文提出了基于因果关系的故障指标根因定位方法, 通过分析不同时序度量指标间的量化因果关系以及不同异常指标在指标间的故障传播情况, 得到最有可能是导致服务实例节点故障的指标根因集合序列.

为了提高指标根因定位的准确性, 并解决软件故障在度量指标间的传播问题, 本文采用了基于特征向量中心性改进的 PageRank 算法, 将指标节点的异常分数作为输入, 同时考虑了影响图中的边的方向及其相关性权重, 将影响关系转换为跨指标节点的故障传播概率, 最终得出每个指标节点的最终异常分数. 基于因果关系的故障指标根因定位方法如图 5 所示, 分为 3 个阶段.

- 首先, 利用多层次影响图中的指标因果层量化指标间的相互作用关系. 同时, 为了后续故障根因定位, 需要对指标间因果关系进行回溯, 将有向边指向相反方向. 根据度量指标间的量化因果关系构建

一个 $n\times n$ 的关联矩阵，作为度量指标间的关联权重. 本文以故障节点 S 为例，如公式(13)所示.

$$\begin{cases} M_{ij}=\dfrac{P(S,I_i,I_j)}{\sum_{k=1}^{m}P(S,I_j,I_k)} \\ M=[M_{ij}] \end{cases} \tag{13}$$

- 然后，对每个服务实例节点的时序指标的异常分数进行计算，并进行归一化，得到异常分数初始值，以此构建一个 $n\times 1$ 的矩阵向量 v，如公式(14)所示.

$$v=[FI(S)_1,FI(S)_2,\ldots,FI(S)_m]^T \tag{14}$$

- 在此基础上，结合 PageRank 算法对每个时序度量指标的异常分数初始值进行计算，如公式(15)所示.

$$v'=dMv+\frac{1-d}{N}U \tag{15}$$

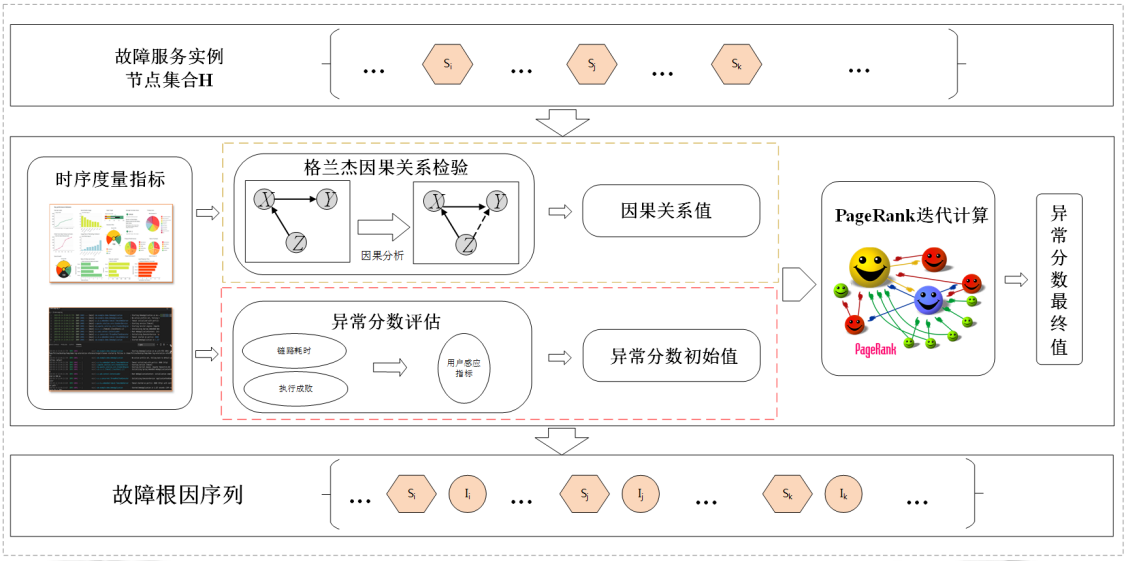


图 5　基于因果关系的故障指标根因定位方法图

此外，本文通过幂次迭代算法将上述 PageRank 算法进行迭代计算，并对结果进行排序，得到对应于每个故障节点的前 10 个故障指标及其异常分数.

最终，将可能引发微服务软件故障的故障节点和故障指标以多元组的形式表示，如公式(16)所示，并依照异常分数的大小有序加入故障根因集合.

$$C_k=(S_i,I_j,F(S),FI(S)_j) \tag{16}$$

具体地，由于微服务软件故障的表征，即服务节点故障和软件用户感应指标的故障，主要是源自故障指标根因的故障，即异常分数越大的故障指标越有可能是引发故障表征的原因. 因此，本文将故障指标的最终异常分数作为判断故障根因的依据，并将上述故障根因集合对故障指标的异常分数大小进行自高到低的排序. 一个故障指标的最终异常分数越大，则该指标越有可能是最终故障根因. 该故障根因集合旨在辅助软件运维人员开展根因定位工作，节省故障排查的时间，提高故障诊断的效率.

4　实验分析

本节介绍本文对多层次影响图故障定位方法开展的实验及结果分析工作. 首先介绍实验的具体设计方法，然后给出实验的实际结果并进行分析讨论.

4.1 实验设计

(1) 基准方法的选择

本文综合分析了现有的微服务软件故障定位方法, 选取了 3 种不同的基准方法, 主要包括 Chen 等人提出的基于因果关系模型的 CauseInfer 方法[13]、Mariani 等人和 Kim 等人分别提出的基于影响关系模型的 Loud 方法[14]和基于随机游走的 MonitorRank 方法[15]. 上述方法介绍以及选取上述方法的主要原因总结如下.

- CauseInfer[13]是针对微服务领域性能问题的根因定位方法, 其主要考虑物理资源、逻辑资源(锁)异常引起的微服务异常问题, 可以自动构造一个两层的层次因果图, 并通过一系列的统计方法, 沿着图中的因果路径推断性能问题的原因. 故障关系模型主要分为调用图、因果图和影响图这 3 类. 因果图和影响图是目前应用最广泛的、用于微服务故障定位的故障关系模型, 在故障根因定位的准确性和性能上有较大优势. 因此, 本文选择了基于故障因果图和影响图的故障定位方法 CauseInfer 进行对比.
- Loud[14]将机器学习与图中心性算法相结合, 该方法依赖于关键性能指标(KPI), 即通常在运行系统上收集的指标, 利用机器学习来检测 KPI 中的异常并揭示它们之间的因果关系, 利用基于中心性指标的算法来补充机器学习, 以定位产生和传播异常的故障资源. 本文方法相比于 Loud 更细粒度地分析了度量指标之间的因果关系, 排除了伪因果问题对故障定位结果的影响, 同时对故障影响图模型做出了一定改进.
- MonitorRank[15]是最早使用随机游走的策略定位故障根因服务的方法, 该方法从任何一个怀疑的节点开始入手, 进一步查看其依赖的其他高相关性的节点, 被访问越多的节点越可能是根因.
- PageRank 方法[11]是一种基于网络连接计算节点中心性分数的有效方法, 该算法的基本原理主要通构建随机游走模型的方式, 表示有向图中节点的重要性沿着有向边向周围扩散的现象. PageRank 方法是在随机游走的基础上进行的改进, 作为典型的图中心性计算方法, 相比于随机游走算法更能满足微服务软件复杂结构和不确定环境的要求.

综上, 在软件故障定位领域, 基于迭代深度优先搜索的 CauseInfer[13]、采用 PageRank 算法的 Loud[14]和基于随机游走的 MonitorRank[15]这 3 种方法在模型建模、定位过程等方面具有代表性, 并且能够符合本文的研究目标, 实现细粒度的故障根因定位. 因此, 本文将上述 3 种方法作为本文的对比基准方法, 能够验证 AmazeMap 的有效性和性能, 并保证实验结果的可靠性.

(2) 研究问题

为了验证多层次影响图故障定位方法的有效性, 本文尝试回答以下研究问题.

- RQ1. 本文故障定位有效性如何? 能否检测出软件故障的发生, 并定位故障的节点和指标根因?
- RQ2. 本文故障定位方法的准确性和效率如何? 与现有方法相比, 故障定位的准确性是否得到提高, 所花费的时间开销是否有所减少?

(3) 实验数据集

本文根据如下 3 个原则对数据集进行筛选: 首先, 实验数据集必须来源于微服务系统, 同时包括了部署机器、服务实例和度量指标这 3 个层面的实验数据; 其次, 基准系统或实验数据集必须具有一定的适用性和应用性, 在针对微服务故障的相关研究工作中被广泛引用; 最后, 为了满足对故障数据采集方法有效性和故障定位方法准确性验证的要求, 选择的基准系统必须支持人为故障注入和数据采集工作, 从而能够针对不同类型的微服务软件故障进行重复实验和在同场景下针对现有方法进行对比实验. 因此, 本文选择了基于 Sock-shop 开源基准系统[16]的实验数据集和 AIOps 挑战赛发布的数据集进行实验.

Sock-shop[16]是一个典型的微服务软件基准系统, 模拟电子商务网站的袜子销售流程, 广泛应用于微服务相关研究的测试环节. 不同功能模块用不同编程语言编写, 支持独立开发、部署和数据库存储, 能够有效反映微服务软件系统的特点, 得到了相关研究学者的广泛认同. 同时, 该系统支持故障注入操作, 能够有效反映微服务软件系统在发生软件故障时的真实状态. 该系统采用 SpringBoot[17]、GO Kit[18]和 Node.js[19]进行构建, 主要由 13 个异构技术实现的微服务模块组成, 并通过 HTTP 上的 REST 轻量级通信协议完成服务间交互. 具体

如图 6 所示.

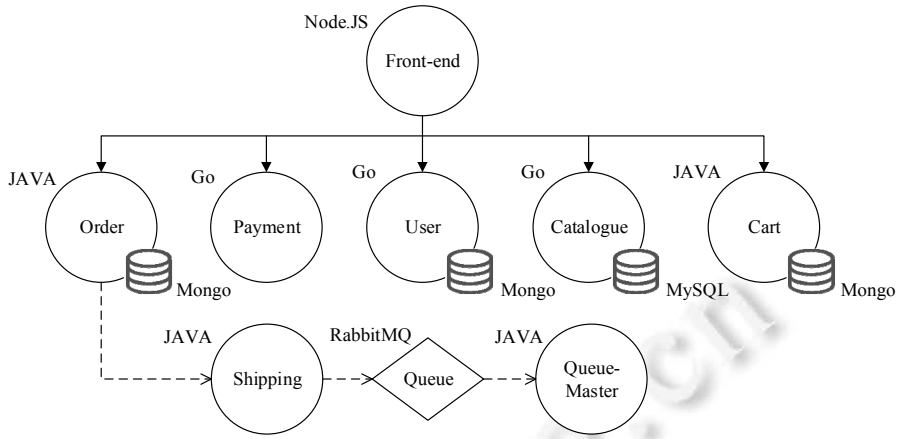


图 6 面向 Sock-shop 基准系统架构图

本文中设置为每 5 s 采集一次度量指标数据, 并存储在相应数据库中. 然后, 采用开源的 AMS 框架对相应功能模块的字节码进行增强, 通过动态插装代理的方式采集可执行链路信息. 本文采用 Chaos Mesh[20]混沌工程测试工具对 Sock-shop 系统进行服务内存泄漏、CPU 占用和网络时延这 3 种类型的故障注入, 从网络、容器和 Pod 层面模拟真实场景下的软件故障. 最终, 本文针对每个故障类型分别随机注入 4 个特定的功能模块的服务实例中, 重复实验 5 次, 得到 60 组实验数据集, 见表 3.

表 3 基于 ChaosMesh 工具对 Sock-shop 故障注入数据集

故障类型	符号表示	解释	注入次数
内存泄漏	memory leak	模拟注入内存泄漏故障	20
CPU 占用	CPU fault	模拟容器 CPU 压力	20
网络延迟	network delay	模拟注入网络延迟故障	20

然后, 通过人工标注的方式获取验证数据集, 记录每次故障注入测试的时间, 获取到基于 Sock-shop 开源基准系统数据集. 详细数据集指标见表 4.

表 4 Sock-shop 故障注入数据集指标表

维度	符号	定义
容器	I_{reum}	container_cpu_used: 容器 CPU 使用率
	I_{ctup}	container_thread_used_pct: 容器线程使用率
	I_{ocup}	CPU_util_pct: 系统 CPU 使用率
内存	I_{omup}	Memory_used_pct: 系统内存使用率
	I_{odiu}	Disk_io_util: 系统 IO 繁忙度
	I_{reum}	used_memory: Redis 内存使用量
	I_{rumr}	used_memory_rss: Redis 分配的内存总量
	I_{rtcr}	total_connections_received: Redis 连接请求数
网络	I_{nsp}	Sent_queue: 网络发送队列数
	I_{nrep}	Received_errors_packets: 网络接收错误包数
业务	I_{rt}	服务请求响应时间
	I_{tp}	系统在单位时间内处理的请求数-吞吐量
	I_{err}	一段时间服务调用失败占总调用数的比值

AIOps 挑战赛数据集是针对某大型的电子商城在私有云环境下的微服务软件运行数据, 是专门用于微服务软件系统故障发现和根因定位的实验数据集. 该微服务软件系统部署搭建在 22 个机器节点, 并集成了 Oracle[21]、Redis[22]和 Docker[23]组件, 每天的平均数据量约为 1.2 GB, 所产生的数据集提供了包括业务功能、调用链、容器、中间件、主机节点和数据集的指标数据以及相应微服务软件系统的部署文档. 该数据集有如

下 3 个特点: 其一, 该数据集包括了服务调用指标、业务功能指标和平台性能指标(包括数据库、中间件、容器等), 有效指标达 100 个以上, 能够全面反映微服务软件系统的运行状态, 满足了本文所提的故障定位方法中对于度量指标和可执行链路信息的数据需求; 其二, AIOps 挑战赛官方通过分批次实时故障注入的方式获取故障数据, 注入时间为每天的 0:00–6:00, 共计 6 h, 对故障注入的时刻、持续时间进行记录, 并确保任意时刻只存在唯一故障; 其三, 该数据集来源于大规模微服务软件系统在真实场景下的故障注入测试数据, 故障的类型多样, 数据规模庞大, 在学术界具有一定的影响力.

本文从 2020 年 4 月 11 日–2020 年 5 月 31 日期间随机抽取了 15 日软件运行数据, 并将 CPU 占用、网络延迟、网络丢包以及数据库连接失败这 4 种故障类型作为本文的故障定位方法对比实验的故障类型, 见表 5, 最终得到了 15 组实验数据集. 此外, 由于一个大型微服务软件系统涉及大量的组件和技术栈, 导致该数据集中存在大量的指标. 为了降低故障定位方法的时间和资源开销, 本文从 AIOps 数据集中筛选出了 31 个重要指标, 具体包括用户感应指标、中间关联指标、环境部署指标这 3 种类型, 涵盖了操作系统、容器、数据库、Redis 中间件等多个层次的指标, 其中一部分重要指标见表 6.

表 5 面向 AIOps 挑战赛数据集故障类型表

故障类型	符号表示	解释
CPU 占用	CPU fault	模拟容器 CPU 压力
网络延迟	network delay	模拟注入网络延迟故障
网络丢包	network loss	模拟注入网络延迟故障
数据库连接失败	db connection limit	模拟数据库连接限制问题

表 6 面向 AIOps 挑战赛数据集指标表

类型	维度	符号	定义
环境部署指标	容器	I_{reum}	container_cpu_used: 容器 CPU 使用率
		I_{ctup}	container_thread_used_pct: 容器线程使用率
	操作系统	I_{omup}	Memory_used_pct: 系统内存使用率
		I_{odiu}	Disk_io_util: 系统 IO 繁忙度
		I_{ocup}	CPU_util_pct: 系统 CPU 使用率
	网络	I_{nsp}	Sent_queue: 网络发送队列数
		I_{nrep}	Received_errors_packets: 网络接收错误包数
中间关联指标	数据库	I_{pup}	Proc_Used_Pct: 数据库连接使用百分比
		I_{trt}	tnsping_result_time: 数据库响应时间
		I_{dbps}	TPS_Per_Sec: 数据库每秒事务数
		I_{dbsc}	Sess_Connect: 数据库连接会话数
	中间件	I_{reum}	used_memory: Redis 内存使用量
		I_{rumr}	used_memory_rss: Redis 分配的内存总量
		I_{rtcr}	total_connections_received: Redis 连接请求数
用户感应指标	业务	I_{rt}	服务请求响应时间
		I_{tp}	系统在单位时间内处理的请求数-吞吐量
		I_{err}	一段时间服务调用失败占总调用数的比值

(1) 评测指标

- 精确率(*precision*): 在某时刻 T 下, 故障定位根因集合中正确定位率, 即查准率.
- 召回率(*recall*): 正确定位出故障根因的已检测故障事件占所有检测出故障事件的比例, 即查全率.
- $F1$ 值: 精确率和召回率的调和平均, 可以用来综合评价故障定位方法.
- Recall of Top-k (*PR@K*): 用于表示故障根因降序集合中, 前 K 个根因结果包含真实根本原因的概率.
- 定位效率: 表示从故障发生时刻到定位出具体的故障根因之间的耗时.

(2) 运行环境配置

本文的实验环境由集成配置信息、Sock-shop 系统集群硬件部署环境、集成的软件配置信息组成. 其中, 实验环境的集成配置主要采用的编程语言为 JavA, IDE 版本为 IDEA 2018.3.6. 系统集群部署环境由 1 台 Master 机器节点(i5-7500, 16 GB)和 4 台工作节点(i5-7500, 8 GB)组成.

4.2 算法有效性测试

(1) 定位有效性测试(RQ1)

本文针对 Sock-shop 系统数据集对 AmazeMap 提出的故障检测和根因定位进行了有效性验证. 首先, 根据数据集中的故障数据, 构建多层次故障影响图; 然后, 计算出每条执行链路和服务实例节点的异常分数, 并根据动态阈值检测出软件故障; 最后, 采用基于影响图的故障定位方法实现对故障服务节点和指标节点的根因定位.

首先, 本文根据 Sock-shop 系统数据集中的执行链路数据、时序度量指标数据和系统部署信息完成多层次故障影响图建模. 本文通过多次实验, 将故障数据采集的时序时间窗口限制在 5 min (其中, 故障注入的持续时间为 60 s, 采集故障注入前后各 2 min 的软件运行数据), 采集间隔为 1 s, 从中筛选采集到共计可执行链路 79 条, 服务实例总数 32 个, 对应于每条链路的链路耗时以及对应每个服务实例节点的服务响应时间、CPU 使用率等 31 个时序度量指标的数据变化.

然后, 本文通过时序度量指标因果关系量化计算方法, 计算在时间窗口下时序度量指标间的关联关系. 本文以在注入网络延迟后 45 s 后特定故障服务实例节点 Order2 为例展开介绍, 并对产生明显波动的重要指标间的量化因果关系值进行展示, 具体见表 7.

表 7 度量指标因果关系值对应表

度量指标类别	I_{bps}	I_{dbrt}	I_{cpu}	I_{tps}	I_{tp}	I_{rt}
I_{bps} (网络传输速率)	0	0.813	0.639	0.607	0.846	0.919
I_{dbrt} (数据库响应时间)	0.315	0	0.326	0.453	0.463	0.445
I_{cpu} (容器 CPU 使用率)	0.120	0.237	0	0.727	0.694	0.879
I_{tps} (数据库每秒事务数)	0	0	0.432	0	0.534	0.624
I_{tp} (单位时间处理请求数)	0.433	0.598	0.621	0.219	0	0.796
I_{rt} (服务响应时间)	0.511	0.257	0	0	0.833	0

在此基础上, 综合分析服务部署状态、服务调用信息、时序度量指标及影响关系, 构建多层次故障影响图模型. 以上述故障注入后 45 s 时刻下涉及服务实例 Order2 的执行链路为例, 在服务实例节点 Order2 涉及 3 条可执行链路, 其中, 网络传输速率、网络发送队列数、容器 CPU 使用率等指标均有可能是导致服务实例节点 Order2 响应时间异常的根本原因.

然后, 为了准确检测出软件故障, 并缩小后续故障根因定位范围, 本文针对当前采集到 79 条执行链路的链路耗时, 所在故障链路上和与故障节点部署在同一机器节点上的服务实例的用户感应指标, 即吞吐量、错误率和响应时间进行异常分数评估, 并根据不同指标的预测偏差值计算判定故障的动态阈值, 从而完成故障检测过程, 进而对故障影响图进行精化. 本文以服务实例节点 Order2 和 Catalogue1 以及其所在的链路 Trace41 的故障状态为例展示故障检测结果, 具体见表 8.

表 8 链路和服务实例节点故障检测判定表

所属位置	指标	异常分数	阈值	是否故障
Trace41	链路耗时	0.562	0.179	是
Order2	响应时间	0.771	0.223	是
	吞吐量	0.514	0.235	是
	错误率	0.102	0.192	否
Catalogue1	响应时间	0.452	0.159	是
	吞吐量	0.317	0.244	是
	错误率	0.096	0.181	否

其中, 服务实例 Order2 响应时间的动态阈值计算如图 7 所示. 正常节点率在 0.3 以下时随阈值提高而提高, 阈值达到一定界限之后逐渐趋于稳定; 最大距离表示的是正常节点率到 $y=x$ 直线的垂直距离, 用于找出最大阈值点. 该曲线的局部最大值为最优动态阈值, 即 $x=0.233$, 此时正确节点率曲线达到最大曲率值, 能够有效检测出故障服务节点.

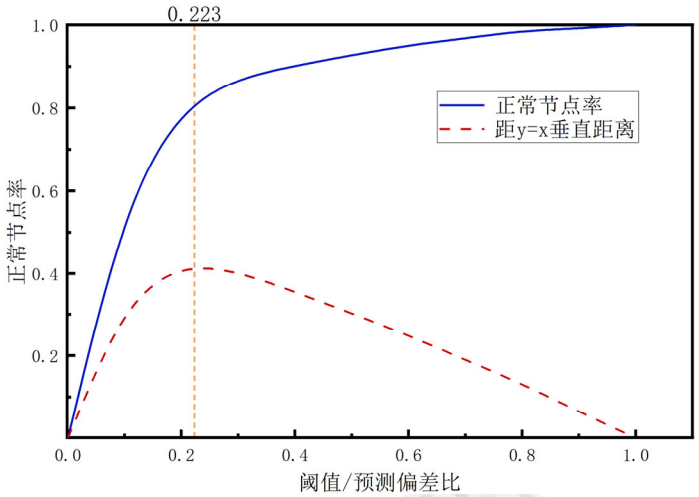
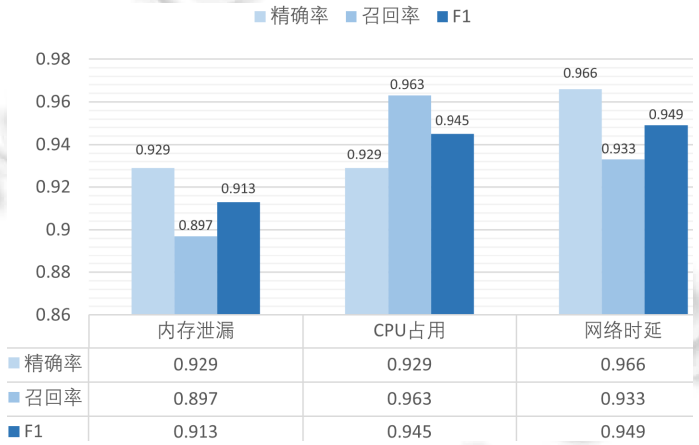


图 7 动态阈值计算曲线图

最后，在精化故障影响图的基础上，本文针对服务内存泄漏、CPU 占用和网络时延这 3 类故障进行实验，通过基于 PageRank 的根因定位方法实现对故障节点和指标根因进行定位，输出根因概率最大的前 10 个根因序列集合. 同时，将该根因序列集合与故障注入的故障类型和故障服务节点位置进行对比，并选取精确值、召回率和 *F*1 评价指标作为故障定位方法的有效性验证，具体如图 8 所示.



	内存泄漏	CPU占用	网络时延
精确率	0.929	0.929	0.966
召回率	0.897	0.963	0.933
F1	0.913	0.945	0.949

图 8 面向 Sock-shop 数据集的故障定位方法有效性评估结果

本文所提的故障定位方法针对 Sock-shop 数据集上的 3 种故障都有较好的定位效果. 其中，定位结果的精确率和 *F*1 值均在 0.9 以上，并通过根因定位方法有效定位故障服务实例节点和指标根因. 而在召回率方面，由于微服务实例的内存泄漏故障涉及容器、操作系统等多个层面的影响，内部影响关系对比 CPU 占用和网络时延故障而言更为复杂，因此在召回率方面仅达到 0.897，但其误差在可接受范围之内，仍能够支持微服务故障定位输出正确根因结果.

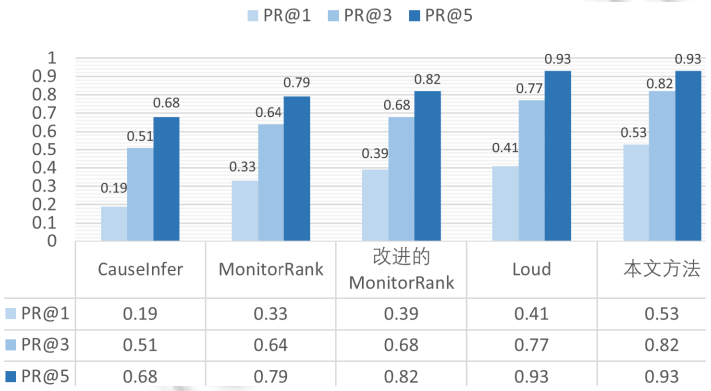
综上，通过上述实验步骤，可以判断本文提出的基于故障影响图的软件故障定位方法能够有效定位微服务软件故障，从而帮助微服务软件运维人员快速准确地定位软件故障问题.

(2) 故障定位方法准确性和效率验证(RQ2)

本文方法在 Sock-shop 数据集以及 AIOps 挑战赛数据集上与现有方法的准确性和效率进行验证. 选用 *PR@K* 和定位耗时作为故障定位方法准确性和效率的评价指标. 此外，由于 MonitorRank 方法中对于微服务

软件故障的检测效果并不好, 而本节的对比实验旨在比较根本原因定位的效果, 因此, 为了提高本次对比实验的合理性, 本文将基于本文故障检测结果的改进 MonitorRank 方法也作为对比方法进行实验.

首先, 本文基于 Sock-shop 开源基准系统数据集中关于内存泄漏、CPU 占用和网络时延这 3 类故障的数据开展故障定位方法准确性对比验证, 得到每个方法在 *PR*@1、*PR*@3 和 *PR*@5 的平均性能指标. 具体结果如图 9 所示, 文中选取的故障定位方法在 Sock-shop 数据集上均能够准确定位故障根因, 且随着评价指标中 *K* 值的提高, 故障定位的准确性也逐渐提高. 与 Loud 方法相比, 本文方法在 *PR*@5 上一致, 在 *PR*@1 和 *PR*@3 上分别提高了 12%和 5%. 这是由于本文方法对于服务和指标故障节点的定位排除了伪因果问题的影响, 在 *K* 值较低的情况下的准确性优于 Loud 方法. 而与其他基准方法相比, 本文方法在 *PR*@1、*PR*@3 和 *PR*@5 的定位结果分别提高了 14%、14%和 11%.



	CauseInfer	MonitorRank	改进的 MonitorRank	Loud	本文方法
PR@1	0.19	0.33	0.39	0.41	0.53
PR@3	0.51	0.64	0.68	0.77	0.82
PR@5	0.68	0.79	0.82	0.93	0.93

图 9　基于 Sock-shop 数据集的故障定位方法准确性评估

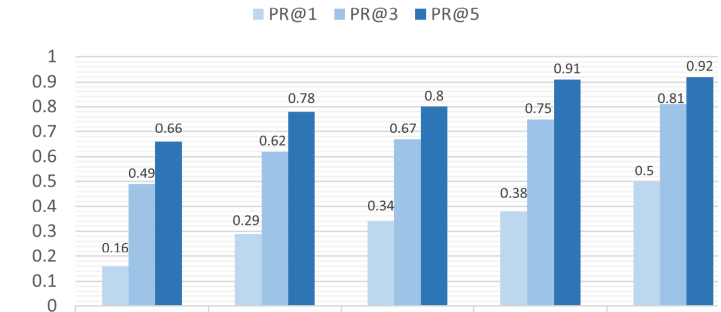
同时, 在 *PR*@1 方面, 本文参与实验方法输出的排名第一的根因序列与真实根因均存在一定误差, 故障定位的准确性都不高, 尤其是 CauseInfer 方法的 *PR*@1 值仅为 19%. 这是由于仅通过 PC 算法分析故障指标间关联关系, 从而构建故障因果图模型的性能较差. 改进的基于随机游走的 MonitorRank 方法相较于原 MonitorRank 方法, 在 *PR*@1、*PR*@3 和 *PR*@5 值分别提高了 6%、4%、3%. 但是该方法最终故障定位效果还是有所欠缺, 这是由于 MonitorRank 方法过于依赖故障图模型, 当某个正常微服务出现在多条故障链路中, 会提高该微服务的异常分数, 从而导致故障定位的失败. 相反, 本文所提方法与 Loud 算法在此次实验中均具有出色表现. 这是由于这两种方法都通过格兰杰因果关系检验, 进一步筛查了可能的故障根因, 排除了相关正常微服务的影响. 此外, 本文方法通过对伪因果关系的排查和对故障服务和指标节点的筛查, 在故障定位的准确性上更为出色.

然后, 本文基于 AIOps 挑战赛数据集中关于 CPU 占用、网络延迟、网络丢包以及数据库连接失败这 4 种故障类型的数据进行对比实验, 具体结果如图 10 所示. 可以看出, 本文方法和对比基准方法在 AIOps 挑战赛数据集中同样具有良好的故障定位效果. 其中, 本文方法与其他基准方法相比, 在 *PR*@1、*PR*@3 和 *PR*@5 这 3 类评价指标上分别提高了 12%、6%和 1%, 所选用的实验方法随 *K* 值的变化趋势和基于 Sock-shop 数据集的故障定位对比实验变化趋势相似.

此外, 通过对比上述两个基准系统的对比实验结果, 可以发现: 由于 AIOps 数据集涉及的度量指标更多, 基准微服务系统更为复杂, 故障类型更为多样, 导致基于 AIOps 的故障定位方法的准确性均有一定程度的降低. 通过对比可以看出: 相比于其他方法, 本文方法面向不同微服务系统的效果稳定性更好, 在 *PR*@1, *PR*@3 和 *PR*@5 这 3 类评价指标上分别仅降低了 3%、1%和 1%. 由于微服务软件系统自身复杂的结构和不确定性环境因素, 本文认为, 本文方法针对复杂系统的误差仍在可接受范围之内.

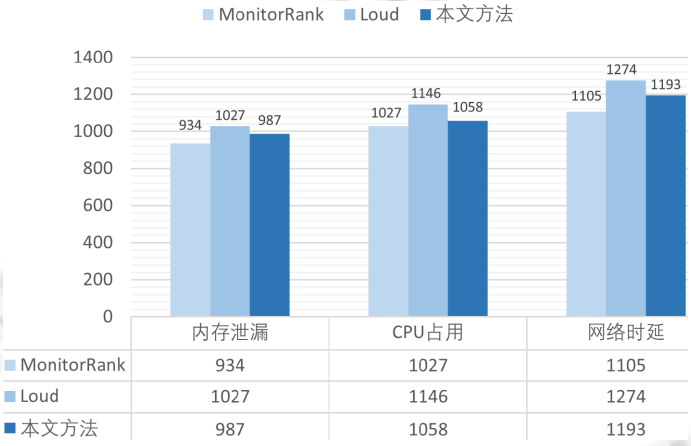
最后, 在故障定位准确性实验的基础上, 本文测试了故障定位准确性较好的 MonitorRank、Loud 和本文方法的故障定位耗时, 通过计算故障注入时刻到获得故障定位结果时刻的差值, 作为该方法的定位耗时, 并

以此用于评估故障定位方法的效率, 具体如图 11 所示.



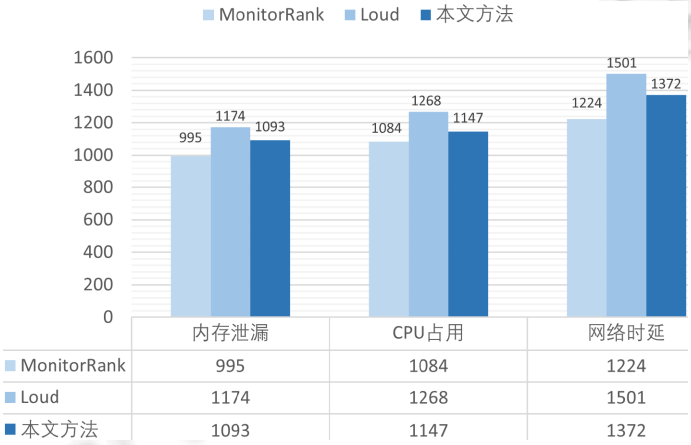
	CauseInfer	MonitorRank	改进的 MonitorRank	Loud	本文方法
PR@1	0.16	0.29	0.34	0.38	0.5
PR@3	0.49	0.62	0.67	0.75	0.81
PR@5	0.66	0.78	0.8	0.91	0.92

图 10 基于 AIOps 数据集的故障定位方法准确性评估



	内存泄漏	CPU占用	网络时延
MonitorRank	934	1027	1105
Loud	1027	1146	1274
本文方法	987	1058	1193

(a) 基于 Sock-shop 数据集效率对比实验评估结果



	内存泄漏	CPU占用	网络时延
MonitorRank	995	1084	1224
Loud	1174	1268	1501
本文方法	1093	1147	1372

(b) 基于 AIOps 挑战赛数据集效率对比实验评估结果

图 11 故障定位方法效率对比实验评估

根据上述效率对比实验的评估结果, 基于 AIOps 数据集的故障定位耗时普遍高于基于 Sock-shop 数据集的故障定位耗时. 这是由于 AIOps 数据集所选取的微服务基准系统规模庞大, 结构复杂, 考虑的度量指标全面所导致的. 同时, 针对不同的故障类型, 故障定位耗时也不同. 可以看出: 由于 MonitorRank 方法仅考虑服务间的因果依赖关系, 在方法耗时上最少, 但方法的准确性较差. 而对比于在 *PR*@5 准确性上相似的 Loud 方法, 本文的故障定位方法耗时更少. 这是由于本文方法在故障检测和故障节点根因定位环节做了两次筛选, 缩小了故障定位的范围, 从而在一定程度上降低了定位耗时, 提高了故障定位的效率.

综上, 本文针对两个数据集开展故障定位方法的准确性和效率验证, 并进行了多次实验. 该实验结果说明: AmazeMap 相比于其他现有基准方法, 在准确性和效率有更好的表现, 能够快速准确地定位故障服务和指标根因, 有效帮助运维人员维护微服务软件系统的正常运行, 快速定位故障.

4.3 消融实验设置

为了验证 AmazeMap 中每个组件的贡献以及重要程度, 分别构建两种 AmazeMap 的变体, 并设计一系列实验来比较其性能.

(1) AmazeMap w/o 指标时序层. 为了验证指标时序层在细粒度根因定位过程中的重要性, 将指标时序层从 AmazeMap 中删除.

(2) AmazeMap w/o 指标因果层. 为了衡量指标因果层在细粒度根因定位过程中发挥的作用, 将指标因果层从 AmazeMap 中去除. 因服务调用层为细粒度根因定位基础, 在确定服务集合后进一步定位指标级根因, 故上述实验设置未针对服务调用层进行设计.

实验结果如图 12 所示, 分别列出上述两种变体在两种数据集上的 *PR*@1、*PR*@3、*PR*@5.

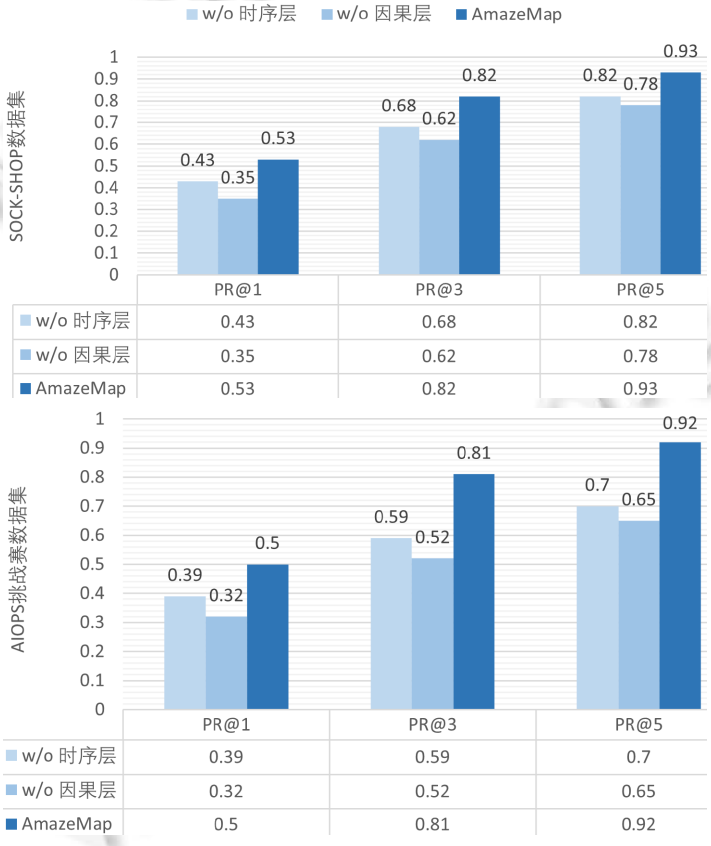


图 12 AmazeMap 与不同模型变体的 *PR*@1、*PR*@3、*PR*@5

当指标时序层被删除后, *PR*@1、*PR*@3、*PR*@5 均有不同程度降低. 实验结果表明: 对于指标这类时间序列, 指标时序层能够建模指标时序关系, 捕捉指标随时间变化的趋势, 从而减少误报. 当指标因果层被删除后, 各项指标都呈现不同程度的下降趋势, 且相较于去除指标因果层下降趋势更大, 表明在 AmazeMap 中, 指标因果层相较于指标时序层更重要, 指标时序更多从时间维度考虑, 而指标因果关系能够较好地描述不同指标间的关联关系. 此外, 由于 AIOps 挑战赛数据集相比于 Sock-shop 数据集更加复杂, 故在 Sock-shop 数据集上的结果总体优于 AIOps 挑战赛数据集的结果.

5 相关工作

现有的故障定位方法大部分应用于单体软件和分布式软件, 关注于在运行环境稳定状态下的软件系统的程序内部逻辑故障, 无法适用于在不确定环境下的微服务软件系统的复杂故障定位[12,14]. 此外, 现有方法较少从时空融合角度处理运维数据, 忽略微服务系统的时间和空间特性. 目前, 针对微服务软件故障定位的方法大体分为故障检测和根因定位两个部分. 针对故障检测和根因定位的有关工作较为分散, 两者无法进行有效结合, 从而导致微服务软件故障定位过程更为复杂, 造成了极大的阻碍. 因此, 本文对国内外故障检测和根因定位方法展开调研与分析, 旨在分析当前在微服务故障定位领域所面临的问题和挑战.

5.1 微服务故障检测方法现状

本文根据国内外微服务故障检测方法所使用的输入数据的类型差别, 将其分为基于日志的故障检测、基于链路的故障检测和基于度量指标故障检测这 3 类.

基于日志的故障检测方法通常采用无监督机器学习算法,主要通过学习采集到的无故障状态下的软件运行日志, 通过判断在线状态下新日志信息是否偏离基准模型来检测是否发生软件故障[24,25]. 上述方法立足于以相似顺序能够记录相似的事件. 然而, 微服务软件频繁变更的软件需求可能会通过引入新服务来取代旧服务来完成功能的变更[26]. 因此, 上述故障检测方法无法保证准确性.

基于链路的故障检测方法主要是通过采集到的微服务软件可执行链路, 利用机器学习[27,28]或链路对比技术[29,30]对故障进行检测. 与上述机器学习方法不同的是, 复旦大学彭鑫教授团队提出一种 MEPFL 模型[31], 通过将一个微服务应用程序与一组模拟用户请求的自动化测试用例作为输入, 并通过监督学习算法分析不同类型的故障对应的故障表征来完成在线故障检测. 张攀等人[32]提出一种基于自然语言处理与双向长短期记忆(BiLSTM)网络的微服务调用链异常检测方法 MicroTrace, 利用事件进行检测.

基于度量指标故障检测方法大部分通过安装代理的方法实时采集软件运行时指标, 并采用机器学习算法对相关指标进行模型训练[14], 以此来确定微服务软件在应用级和服务级的故障问题. 然而, 上述都是在离线状态下训练基准模型, 不会随着时间变化而改变, 无法很好地适用于微服务软件系统. 而 CloudRanger 方法[8]能够通过分析历史监测数据, 预测当前响应时间. 与上述方法不同的是, Hora 模型[33,34]通过将贝叶斯网络模型关联到一个数据采集器来完成在线故障检测. 此外, CauseInfer 方法[7,13]通过比较 KPI 指标与软件系统的 SLO 来检测软件性能故障; Zang 等人则是结合心跳检测机制检测服务级功能故障[35]. MicroRCA 方法[2,36]针对特定的 KPI 指标, 应用于 Kubernetes 部署的微服务软件系统, 并结合 Istio[37]和 Prometheus 组件完成指标采集.

综上所述, 现有大部分基于机器学习的故障检测方法都依赖于微服务软件系统在运行过程采集到的日志、链路和度量指标数据, 通过训练构建基准模型作为判断软件故障的依据. 由于微服务软件系统运行条件和服务模块频繁变更的需求, 上述方法难以解决软件变化导致基准模型不准确的问题. 同时, 还有部分方法只单独针对性能故障或服务故障, 在故障检测的粒度上存在局限性.

5.2 微服务根因定位方法现状

现有根因定位方法主要是在传统方法的基础上, 针对微服务软件特点, 通过将日志、链路和度量指标作为输入, 构建相关模型的方式对此类输入数据进行分析, 如调用图和影响图等, 从而完成根因定位过程. 本文将针对此过程中构建的不同故障相关图模型类型展开分析.

基于调用图的故障根因定位方法主要是通过采集软件运行时服务间相关调用的执行轨迹构建关系模型, 并利用轨迹差异对比[5]或树编辑距离等方法完成故障根因定位. 例如, CloudDiag 工具[38]利用变异系数[39]和基于稳健主成分分析[40]识别出故障根本原因. MicroRank 方法[41]采用 PageRank 算法计算每个服务在整个调用图模型中的中心性, 获得每个服务实例的异常分数. 然而, 由于基于调用图的故障根因定位仅能定位服务级故障, 无法细粒度定位软故障根因, MA 团队[6]通过深入挖掘服务间的依赖关系, 构建故障依赖图模型.

基于因果图的故障定位方法通常利用度量指标与对应服务模型的相关性, 通过量化计算不同指标间的因果关系, 大多采用 PC 算法[42]完成因果图建模过程. 在此基础上, 结合随机游走[43−46]或优先遍历方法[3,7,13,47]访问因果图, 从而确定软件故障的根本原因. 此外, Meng 团队[48]结合软件故障传播时延构建因果图, 通过随机遍历的方式确定故障指标的根因.

基于影响图的故障定位方法则是通过考虑故障传播、依赖关系和指标波动等对目标系统的影响, 分析量化不同故障表征、故障影响因素和故障根因间的关联关系, 进而构建故障影响图. 例如, CloudRanger 工具[8]用数据驱动的影响图代替调用图定位故障的传播模型. MonitorRank 方法[15]和 MicroHECL 方法[49]都考虑了基于服务交互的故障传播问题. MicroRCA[36]和 MicroDiag 方法[50]通过不断采集并检测 SLO 指标判断故障, 并将故障与度量指标相关联. 此外, 还有一部分故障根因定位方法是通过统计分析方法[51−53]直接处理微服务监控数据, 从而定位故障根因.

大部分针对微服务软件的根因定位方法都是通过利用服务调用信息、性能指标、微服务部署等数据构建调用图、因果图和影响图模型, 并采用随机游走算法、统计分析算法等进行故障根因定位, 最终输出多个可能的根本原因. 可以看出, 上述方法严重依赖于故障模型的质量, 对于故障模型的全面性和科学性有很高的要求. 其中, 调用图模型和因果图模型分别针对服务调用关系和故障因果关系对微服务软件系统进行故障模型构建, 但不能全面且准确地定位服务故障和性能故障的根因位置. 而故障影响图模型作为目前较为热门的故障根因定位模型, 能够有效结合调用图和因果图模型的特点, 从故障表征、故障影响因素和故障根因的关系出发, 忽略复杂的系统业务逻辑, 提高微服务软件故障根因定位的准确性和全面性.

6 结论与展望

在不确定环境下, 能够准确定位软件故障是确保大规模微服务软件系统稳定运行的关键因素之一. 因此, 本文创新性地提出了 AmazeMap, 包括多层次故障影响图建模方法和基于多层次故障影响图的微服务故障定位方法. AmazeMap 通过关联服务调用层、基础设施层、时序指标层, 从时间和空间角度挖掘数据间关联关系, 并根据微服务架构软件实际特点设计了节点构建方法和基于格兰杰因果关系检验方法量化计算指标间的作用关系, 整合上述所有信息构成多层次故障影响图, 为故障定位提供模型基础. 在多层次故障影响图模型的支持下, 利用动态阈值计算和图中心性方法快速准确地检测和细粒度定位软件故障, 从故障链路、故障服务和指标根因这 3 个层次对故障进行层层筛查, 缩小故障定位的范围, 减少时间开销, 最终从时序度量指标中细粒度定位故障根因. 此外, 通过对比验证了本文方法在故障定位的准确性和效率上有一定优势, 能够帮助运维人员快速进行故障定位, 提高运维质量.

为了进一步提高微服务故障定位的准确性, 本文仍存在值得继续探讨的工作: (1) 本文后续考虑对跨服务节点上的度量指标作用关系进行建模, 并协调好故障关系建模时间开销和后续故障定位的准确性; (2) 当前, 复杂微服务软件系统不确定性增强, 如何计算故障服务节点和异常分数间的真实关联关系, 是未来研究工作的重心; (3) 目前, 微服务数据集较少, 希望能够从开源项目和商业项目中搜集更多的数据集, 以验证本文所得出的实验结论是否具有一般性.

References:

[1] Bogner J, Zimmermann A. Towards integrating microservices with adaptable enterprise architecture. In: Proc. of the 20th IEEE Int'l Enterprise Distributed Object Computing Workshop (EDOCW). 2016. 1−6. [doi: 10.1109/EDOCW.2016.7584392]

[2] Wu L, Bogatinovski J, Nedelkoski S, Tordsson J, Kao O. Performance diagnosis in cloud microservices using deep learning. In: Hacid H, Outay F, Paik H, Alloum A, Petrocchi M, Bouadjenek MR, Beheshti A, Liu X, Maaradji A, eds. Proc. of the Service-Oriented Computing Workshops (ICSOC 2020). Cham: Springer, 2021. 85−96.

[3] Lin J, Chen P, Zheng Z. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Pahl C, Vukovic M, Yin J, Yu Q, eds. Proc. of the Service-oriented Computing. Cham: Springer, 2018. 3−20.

[4] Nandi A, Mandal A, Atreja S, Dasgupta GB, Bhattacharya S. Anomaly detection using program control flow graph mining from execution logs. In: Proc. of the 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. New York: ACM, 2016. 215−224. [doi: 10.1145/2939672.2939712]

[5] Wang ZY, Wang T, Zhang WB, Chen NJ, Zuo C. Fault diagnosis for microservices with execution trace monitoring. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1435−1454 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5223.htm [doi: 10.13328/j.cnki.jos.005223]

[6] Ma SP, Fan CY, Chuang Y, Lee WT, Lee SJ, Hsueh NL. Using service dependency graph to analyze and test microservices. In: Proc. of the 42nd IEEE Annual Computer Software and Applications Conf. (COMPSAC). 2018. 81−86.

[7] Chen P, Qi Y, Zheng P, Hou D. *CauseInfer*: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: Proc. of the IEEE Conf. on Computer Communications (IEEE INFOCOM 2014). 2014. 1887−1895.

[8] Wang P, Xu J, Ma M, Lin W, Pan D, Wang Y, Chen P. CloudRanger: Root cause identification for cloud native systems. In: Proc. of the 18th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGRID). 2018. 492−502.

[9] Gill SS, Buyya R. Failure management for reliable cloud computing: A taxonomy, model, and future directions. Computing in Science & Engineering, 2020, 22(3): 52−63.

[10] Aguilera MK, Chen W, Toueg S. Failure detection and consensus in the crash-recovery model. Distributed Computing, 2000, 13(2): 99−125.

[11] Langville AN, Meyer CD. A survey of eigenvector methods for Web information retrieval. SIAM Review, 2005, 47(1): 135−161. [doi: 10.1137/S0036144503424786]

[12] Granger CWJ. Some properties of time series data and their use in econometric model specification. Journal of Econometrics, 1981, 16(1): 121−130. [doi: 10.1016/0304-4076(81)90079-8]

[13] Chen P, Qi Y, Hou D. *CauseInfer*: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. IEEE Trans. on Services Computing, 2019, 12(2): 214−230.

[14] Mariani L, Monni C, Pezzé M, Riganelli O, Xin R. Localizing faults in cloud systems. In: Proc. of the 11th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST). 2018. 262−273.

[15] Kim M, Sumbaly R, Shah S. Root cause detection in a service-oriented architecture. In: Proc. of the ACM SIGMETRICS/Int'l Conf. on Measurement and Modeling of Computer Systems. New York: ACM, 2013. 93−104.

[16] 2023. https://github.com/microservices-demo/microservices-demo

[17] 2023. https://spring.io/projects/spring-boot

[18] 2023. https://github.com/go-kit/kit

[19] 2023. https://nodejs.org/en

[20] 2023. https://chaos-mesh.org/website-zh/

[21] 2023. https://www.oracle.com/cn/

[22] 2023. https://github.com/redis/redis

[23] 2023. https://www.docker.com/

[24] Jia T, Chen P, Yang L, Li Y, Meng F, Xu J. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In: Proc. of the 2017 IEEE Int'l Conf. on Web Services (ICWS). 2017. 25−32.

[25] Jia T, Yang L, Chen P, Li Y, Meng F, Xu J. LogSed: Anomaly diagnosis through mining time-weighted control flow graph in logs. In: Proc. of the 10th IEEE Int'l Conf. on Cloud Computing (CLOUD). 2017. 447−455. [doi: 10.1109/CLOUD.2017.64]

[26] Soldani J, Tamburri DA, Heuvel WJVD. The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software, 2018, 146: 215−232.

[27] Rezende DJ, Mohamed S. Variational inference with normalizing flows. In: Proc. of the 32nd Int'l Conf. on Machine Learning, Vol.37. 2015. 1530−1538.

[28] Jin M, Lv A, Zhu Y, Wen Z, Zhong Y, Zhao Z, Wu J, Li H, He H, Chen F. An anomaly detection algorithm for microservice architecture based on robust principal component analysis. IEEE Access, 2020, 8: 226397−226408. [doi: 10.1109/ACCESS.2020.3044610]

[29] Meng L, Ji F, Sun Y, Wang T. Detecting anomalies in microservices with execution trace comparison. Future Generation Computer Systems, 2021, 116: 291−301. [doi: https://doi.org/10.1016/j.future.2020.10.040]

[30] Wang T, Zhang W, Xu J, Gu Z. Workflow-aware automatic fault diagnosis for microservice-based applications with statistics. IEEE Trans. on Network and Service Management, 2020, 17(4): 2350−2363. [doi: 10.1109/TNSM.2020.3022028]

[31] Zhou X, Peng X, Xie T, Sun J, Ji C, Liu D, Xiang Q, He C. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. New York: ACM, 2019. 683−694.

[32] Zhang P, Gao F, Zhou Y, Rao HY, Mao D, Li J. An online real-time anomaly detection method for microservice call chains. Computer Engineering, 2022, 48(11): 161−169 (in Chinese with English abstract). [doi: 10.19678/j.issn.1000-3428.0063817]

[33] Pitakrat T, Okanovic D, Van Hoorn A, Grunske L. An architecture-aware approach to hierarchical online failure prediction. In: Proc. of the 12th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures (QoSA). 2016. 60−69. [doi: 10.1109/QoSA.2016.16]

[34] Pitakrat T, Okanović D, van Hoorn A, Grunske L. Hora: Architecture-aware online failure prediction. Journal of Systems and Software, 2018, 137: 669−685. [doi: 10.1016/j.jss.2017.02.041]

[35] Zang X, Chen W, Zou J, Zhou S, Lisong H, Ruigang L. A fault diagnosis method for microservices based on multi-factor self-adaptive heartbeat detection algorithm. In: Proc. of the 2nd IEEE Conf. on Energy Internet and Energy System Integration (EI2). 2018. 1−6.

[36] Wu L, Tordsson J, Elmroth E, Kao O. MicroRCA: Root cause localization of performance issues in microservices. In: Proc. of the 2020 IEEE/IFIP Network Operations and Management Symp. (NOMS 2020). 2020. 1−9. [doi: 10.1109/NOMS47738.2020.9110353]

[37] Wu FB, Li XY, Pu RQ, Zhang L, Yue HJ. Istio: Research on service governance upgrade of microservice architecture. Network Security Technology and Application, 2022(7): 1−2 (in Chinese with English abstract).

[38] Mi H, Wang H, Zhou Y, Lyu MRT, Cai H. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Trans. on Parallel and Distributed Systems, 2013, 24(6): 1245−1255.

[39] Coefficient of Variation. The Concise Encyclopedia of Statistics. New York: Springer, 2008. 95−96.

[40] Candès EJ, Li X, Ma Y, Wright J. Robust principal component analysis? Journal of the ACM, 2011, 58(3): 1−37.

[41] Yu G, Chen P, Chen H, Guan Z, Huang Z, Jing L, Weng T, Sun X, Li X. MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In: Proc. of the Web Conf. 2021. New York: ACM, 2021. 3087−3098.

[42] Spirtes P, Glymour C, Scheines R. Causation, Prediction, and Search. The MIT Press, 2001. [doi: 10.1007/978-1-4612-2748-9]

[43] Aggarwal P, Gupta A, Mohapatra P, Nagar S, Mandal A, Wang Q, Paradkar A. Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In: Hacid H, Outay F, Paik H, Alloum A, Petrocchi M, Bouadjenek MR, Beheshti A, Liu X, Maaradji A, eds. Proc. of the Service-oriented Computing Workshops (ICSOC 2020). Cham: Springer, 2021. 137−149.

[44] Ma M, Lin W, Pan D, Wang P. MS-Rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications. In: Proc. of the 2019 IEEE Int'l Conf. on Web Services (ICWS). 2019. 60−67.

[45] Ma M, Lin W, Pan D, Wang P. Self-Adaptive root cause diagnosis for large-scale microservice architecture. IEEE Trans. on Services Computing, 2022, 15(3): 1399−1410.

[46] Ma M, Xu J, Wang Y, Chen P, Zhang Z, Wang P. AutoMAP: Diagnose your microservice-based web applications automatically. In: Proc. of the Web Conf. 2020. New York: ACM, 2020. 246−258. [doi: 10.1145/3366423.3380111]

[47] Qiu J, Du Q, Yin K, Zhang S, Qian C. A causality mining and knowledge graph based method of root cause diagnosis for performance anomaly in cloud applications. Applied Sciences, 2020, 10(6): 2166. [doi: 10.3390/app10062166]

[48] Meng Y, Zhang S, Sun Y, Zhang R, Hu Z, Zhang Y, Jia C, Wang Z, Pei D. Localizing failure root causes in a microservice through causality inference. In: Proc. of the 28th IEEE/ACM Int'l Symp. on Quality of Service (IWQoS). 2020. 1−10. [doi: 10.1109/IWQoS49365.2020.9213058]

[49] Liu D, He C, Peng X, Lin F, Zhang C, Gong S, Li Z, Ou J, Wu Z. MicroHECL: High-efficient root cause localization in large-scale microservice systems. In: Proc. of the 43rd Int'l Conf. on Software Engineering: Software Engineering in Practice. Virtual Event: IEEE, 2021. 338−347.

[50] Wu L, Tordsson J, Bogatinovski J, Elmroth E, Kao O. MicroDiag: Fine-grained performance diagnosis for microservice systems. In: Proc. of the 2021 IEEE/ACM Int'l Workshop on Cloud Intelligence (CloudIntelligence). 2021. 31−36. [doi: 10.1109/CloudIntelligence52565.2021.00015]

[51] Shan H, Chen Y, Liu H, Zhang Y, Xiao X, He X, Li M, Ding W. ϵ-Diagnosis: Unsupervised and real-time diagnosis of small-window long-tail latency in large-scale microservice platforms. In: Proc. of the World Wide Web Conf. New York: ACM, 2019. 3215−3222.

[52] Wang L, Zhao N, Chen J, Li P, Zhang W, Sui K. Root-cause metric location for microservice systems via log anomaly detection. In: Proc. of the 2020 IEEE Int'l Conf. on Web Services (ICWS). 2020. 142−150. [doi: 10.1109/ICWS49710.2020.00026]

[53] Kaldor J, Mace J, Bejda M, Gao E, Kuropatwa W, O'Neill J, Ong KW, Schaller B, Shan P, Viscomi B, Venkataraman V, Veeraraghavan K, Song YJ. Canopy: An end-to-end performance tracing and analysis system. In: Proc. of the 26th Symp. on Operating Systems Principles. New York: ACM, 2017. 34−50.

附中文参考文献:

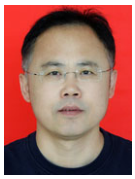
[5] 王子勇, 王焘, 张文博, 陈宁江, 左春. 一种基于执行轨迹监测的微服务故障诊断方法. 软件学报, 2017, 28(6): 1435−1454. http://www.jos.org.cn/1000-9825/5223.htm [doi: 10.13328/j.cnki.jos.005223]

[32] 张攀, 高丰, 周逸, 饶涵宇, 毛东, 李静. 一种在线实时微服务调用链异常检测方法. 计算机工程, 2022, 48(11): 161−169.

[37] 吴封斌, 李笑瑜, 蒲睿强, 张量, 岳洪吉. Istio: 微服务架构服务治理升级研究. 网络安全技术与应用, 2022(7): 1−2.

李亚晓(2000－), 男, 博士生, CCF 学生会员, 主要研究领域为智能化运维, 微服务.

王璐(1991－), 女, 博士, 副教授, CCF 高级会员, 主要研究领域为智能化运维, 微服务与云原生, 软件演化.



李青山(1973－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为国产开源软件, 软件体系结构, 自适应软件演化, 智能化运维, 智能软件工程.

姜宇轩(1999－), 男, 硕士生, 主要研究领域为微服务故障诊断.