软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(2):641−661 [doi: 10.13328/j.cnki.jos.007482] [CSTR: 32375.14.jos.007482]

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

面向大规模在线系统的故障根因变更识别*

余广坝[1], 陈鹏飞[1], 唐锡涛[1], 郑子彬[2]

[1](中山大学 计算机学院, 广东 广州 510006)

[2](中山大学 软件工程学院, 广东 珠海 519082)

通信作者: 陈鹏飞, E-mail: chenpf7@mail.sysu.edu.cn

摘　要: 在大规模在线服务系统中, 为了适应快速变化的用户需求和信息技术如连续集成/交付等, 软件变更频繁发生且呈现上升趋势. 尽管工程师会在软件变更上线之前对新版本进行严格的测试, 但由于测试环境与生产环境之间在负载、规模、用户等方面存在诸多差异, 导致部分隐蔽缺陷未能被及时发现, 随新版本发布带入生产环境, 对系统的可用性和稳定性造成影响. 为了更深入地了解缺陷变更在部署到生产环境后的影响和行为, 基于来自全球大规模即时通信系统微信的真实变更故障数据进行了实证分析, 并得出 5 个关于缺陷变更的关键发现. 基于实证研究的发现和结论, 提出一种轻量级故障根因变更识别方法. 该方法旨在自动化地识别导致变更故障的根因变更, 从而帮助运维工程师完成根因定位和故障修复工作. 为了验证提出的故障根因变更识别方法的有效性, 在微信的生产环境中采集了包含多种类型缺陷变更的真实数据集, 同时还构建一个微服务基准测试系统的模拟变更数据集, 然后对提出的方法进行系统性评估. 实验结果表明, 所提方法在微信生产环境数据集和模拟变更数据上的故障根因变更 Top-3 命中率分别达到 80% 和 84%, 并且故障根因变更识别效果显著优于当前最先进的缺陷变更检测方法. 此外, 从工程实践角度, 系统在处理典型规模故障时内存占用仅为 2.3 GB, 平均分析时延 28.6 s, 满足实际生产环境需求.

关键词: 软件变更; 根因定位; 故障; 在线系统

中图法分类号: TP311

中文引用格式: 余广坝, 陈鹏飞, 唐锡涛, 郑子彬. 面向大规模在线系统的故障根因变更识别. 软件学报, 2026, 37(2): 641–661. http://www.jos.org.cn/1000-9825/7482.htm

英文引用格式: Yu GB, Chen PF, Tang XT, Zheng ZB. Root Cause Change Identification for Failures in Large-scale Online System. Ruan Jian Xue Bao/Journal of Software, 2026, 37(2): 641–661 (in Chinese). http://www.jos.org.cn/1000-9825/7482.htm

Root Cause Change Identification for Failures in Large-scale Online System

YU Guang-Ba[1], CHEN Peng-Fei[1], TANG Xi-Tao[1], ZHENG Zi-Bin[2]

[1](School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China)

[2](School of Software Engineering, Sun Yat-sen University, Zhuhai 519082, China)

Abstract: In large-scale online service systems, software changes occur frequently and are on the rise due to the need to adapt to rapidly changing user demands and information technologies, such as continuous integration and delivery. Although engineers rigorously test new software versions before deployment, some defects may go unnoticed during testing and end up being deployed to the production environment. This primarily occurs because significant differences exist between the testing and production environments in terms of load, scale, and user characteristics. As a result, these defects can impact the system's availability and stability. To better understand the impact and behavior of defective changes after deployment to the production environment, this study conducts an empirical analysis using real change failure data from WeChat, a large-scale global instant messaging system. Five key findings related to defective changes are derived

* 基金项目: 国家重点研发计划 (2024YFB4505904); 国家自然科学基金 (62272495); 广东省基础与应用基础研究基金 (2023B1515020054)
收稿时间: 2024-06-07; 修改时间: 2025-03-25, 2025-05-21; 采用时间: 2025-06-06; jos 在线出版时间: 2025-12-03
CNKI 网络首发时间: 2025-12-04

from this analysis. Based on these empirical findings and conclusions, this study proposes a lightweight root cause change identification method. This method aims to automatically identify the root cause changes that lead to failure, assisting operations and maintenance engineers in root cause localization and trouble shooting efforts. To validate the effectiveness of the proposed method, a real dataset containing various types of defective changes from WeChat's production environment is collected, along with a simulated change dataset based on a microservice benchmark system. A systematic evaluation of the proposed method is then conducted. The experimental results show that the proposed method achieves Top-3 root cause change hit rates of 80% and 84% for the WeChat production environment dataset and simulated change data, respectively, significantly outperforming the state-of-the-art defective change detection methods. Moreover, from an engineering practice perspective, the system uses only 2.3 GB of memory and has an average analysis latency of 28.6 s when processing typical-scale failures, thus meeting the requirements of actual production environments.

Key words: software change; root cause location; failure; online system

1 引 言

在线系统是通过网络连接用户并与用户实时交互的软件系统, 广泛应用于电子商务、社交媒体、金融服务、在线教育等领域, 其普遍性可归因于以下几个方面的优势. 首先, 全球性连接使得在线系统能够跨越空间和时间限制, 实现全球范围内的连接和交互. 其次, 实时性是在线系统的重要特征, 用户能够即时发送请求并获得实时反馈和结果. 第三, 可扩展性使得在线系统能够根据需求动态调整资源和容量, 以适应用户量和数据量的快速增长. 随着微服务、云原生和 Serverless 等技术的不断成熟和应用, 在线系统将继续发展和演进, 为用户带来更多的便利和机遇. 为了适应用户需求和新型信息技术的快速变化, 在线系统开发工程师需要频繁地进行软件变更, 从而引入新功能、改进现有功能或实现创新, 以提供更好的用户体验和满足不断变化的需求. 此外, 进行软件变更还可以有针对性地对系统的性能瓶颈或漏洞进行优化, 提高系统的响应速度、并发处理能力和系统可靠性.

为了提升软件变更的效率, 持续集成和持续交付 (continuous integration and continuous delivery, CI/CD) 作为一种新型的软件开发实践和方法论被提出来. CI/CD 的核心思想是自动化和连续地进行软件构建、测试和发布. 通过使用各种工具和技术, 如版本控制系统、测试自动化框架和部署自动化工具, 开发团队可以实现持续集成和持续交付的目标. 这种方法可以大大减少手动操作和人为错误, 提高软件变更的效率和质量, 同时减少软件变更过程中的失效风险. 得益于 CI/CD 技术的高效, 据 Meta 公司的研究报告, 当前 Meta 在线系统中软件变更的频率可以达到每天上千次[1].

尽管在软件变更上线之前开发工程师会对变更的服务进行严格和周密的测试, 但由于生产环境和测试环境存在多种差异以及更加复杂的组件交互关系, 依然会有部分缺陷变更能通过测试, 并且被部署到生产系统中, 然后导致系统出现故障, 对这些缺陷变更的不当管理可能对用户体验和商业收入产生不利影响. 例如, 在 2021 年, 著名社交软件 Meta 因为错误的软件配置变更, 导致 Meta 及其旗下 Instagram 和 WhatsApp 等应用全网宕机, 停机时间近 7 h. 这 7 h 的应用宕机直接造成超过 9.68 亿美元的损失, 并使 Meta 的市值损失达 643 亿美元[2].

本文将由存在缺陷的软件变更引发的故障称为变更故障. 为了深入了解缺陷变更的影响和行为, 本文基于大规模即时通信系统微信的真实变更故障进行了实证研究, 得出 5 个关键发现 (见第 3 节). 其中在发现 1 中, 本文发现约 66% 的生产环境故障可以归因于缺陷变更, 这进一步证实了软件变更容易导致故障的观点. 在发现 2 中, 本文揭示了与其他故障相比, 在变更引发的故障中具有更严重影响的故障比例更高. 在发现 3 中, 本文统计了不同类型故障根因定位所需时间, 发现确定变更引发的故障的根因比其他故障需要更多时间, 强调了及时发现缺陷变更从而启动变更回滚的迫切性, 缩短故障恢复时间, 提升系统可用性.

考虑到 Google SRE 的"先恢复后根治"原则和 DevOps 责任分离原则[3], 本文聚焦于运维阶段的快速定位导致此次故障的缺陷变更 (即根因变更) 并执行变更回滚来恢复系统 (分钟级), 这与后续开发过程中定位具体的代码或配置缺陷并修复 (小时级) 形成互补. 为了在变更故障发生后快速定位到根因, 当前国内外学者和工业界已经有一些研究工作. 这些研究工作主要集中在服务级别根因分析 (root cause analysis, RCA)[4–7]和缺陷变更检测 (defective change detection, DCD)[8–12]两个方面. 然而, 本文发现现有方法存在两个主要局限性.

● 忽视变更数据. 大多数现有的服务级别根因定位方法[4–7] (如图 1(a)) 通常利用系统的运行时信息 (例如指标和调用链) 作为输入, 但忽略了系统相应的变更数据 (例如灰度变更的流程和变更时间). 因此, 它们往往在服务级别定位根因而无法识别缺陷变更. 在获得服务级别根因定位结果后, 运维工程师需要手动关联相关变更和识别缺陷变更, 导致根因识别时间较长. 因此应该实现自动化的变更级别的根因分析, 以将缺陷变更与变更故障关联起来.

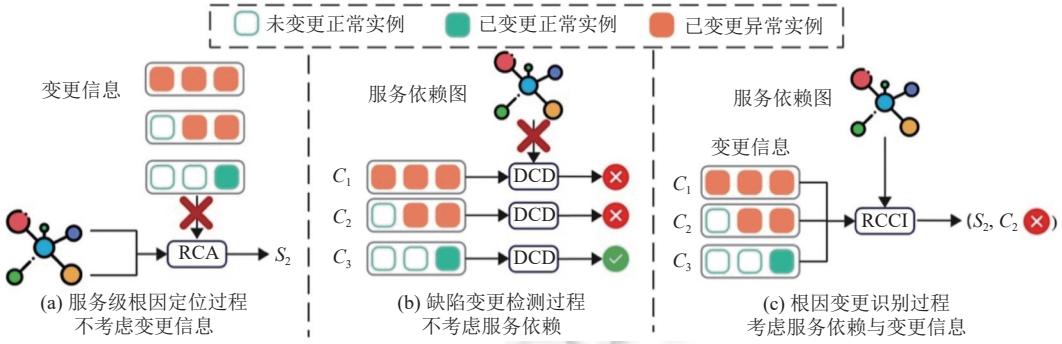

图 1　服务级别根因定位、缺陷变更检测和根因变更识别过程对比

● 忽视异常传播. 缺陷变更检测方法[8–12] (如图 1(b)) 则是以变更数据和运行时信息作为输入, 以诊断单次变更是否是缺陷变更. 但这些缺陷变更检测方法通常不考虑系统中其他服务的变更传播的影响, 只考虑单个服务的单个变更是否异常. 而故障传播在相互依赖的在线服务中经常出现[4], 因此忽视缺陷变更的传播会影响缺陷变更检测算法的诊断, 容易导致出现误报, 进而影响方法的效果. 图 1(b) 的变更 C_1 的异常是由于 C_2 的缺陷变更故障传播导致的, 但缺陷变更检测方法会将 C_1 的变更也判定为缺陷变更.

为了突破现有方法的局限性, 受到实证研究结论的启发, 本文同时考虑服务依赖图信息和变更信息, 提出了变更级根因定位的概念 (如图 1(c) 所示), 本文也称之为根因变更识别 (root cause change identification, RCCI). 需要明确的是, 本文所讨论的“缺陷变更 (defective change)”是指任何引入错误或漏洞的软件修改, 而“根因变更 (root cause change)”则是指导致特定系统故障发生的、经追溯分析确定的根本性缺陷变更. 根因变更识别旨在从诸多变更中精准识别出导致当前故障的根因变更.

为了将根因变更识别的概念应用到真实场景, 本文设计了一种自动化根因变更识别方法, 该方法由既有的告警系统触发, 并包含以下几个步骤: (1) 服务依赖图构建模块构建以告警服务为中心的服务依赖图; (2) 针对服务依赖图上的所有服务, 如果服务在 H h (本文默认为 48 h) 内有变更, 变更依赖构建模块会构建服务内变更之间的依赖关系; (3) 根因变更推断模块则会对每个服务的 H h 内的所有变更计算 KPI 差异得分, 时间衰减得分和空间依赖得分. 然后将这 3 个得分输入根因变更推断模块, 输出一个根因变更得分列表供运维工程师检查. 在理想情况下, 根因变更得分排名第 1 的变更就是导致这次故障的根因变更. 运维工程师可通过回滚或快速修复根因变更使系统恢复正常, 避免对用户的持续影响, 提升系统的可用性.

为了验证提出方法的有效性, 本文基于 Python 实现了根因变更识别方法的原型系统, 并在从微信系统采集的真实变更故障数据集和一个微服务基准测试系统的模拟变更数据集上进行了测试. 实验结果表明, 所提方法在微信生产环境数据集和模拟变更数据集上的 Top-3 命中率 (即根因变更位于推荐列表前 3 位的概率) 分别达到 80% 和 84%, 其识别效果显著优于当前最先进的缺陷变更检测方法. 此外, 从工程实践角度, 系统在处理典型规模故障时内存占用仅为 2.3 GB, 平均分析时延 28.6 s, 满足实际生产环境需求.

综上所述, 本研究的主要贡献如下.

● 基于微信系统生产环境的真实故障数据进行了实证研究, 揭示了缺陷变更对生产环境的影响和相关的行为特征, 并得出 5 个重要发现. 这些发现不仅驱动了本文的研究, 还能够给未来的研究带来启发 (第 3 节).

● 提出了一种根因变更识别方法. 在变更故障发生时, 它能够更准确、更快速地从大量软件变更中识别出导

致故障的根因变更. 这有助于运维工程师关注根因变更并采取适当的措施, 以避免进一步的服务中断 (第 4 节).

● 实现了根因变更识别算法的原型, 并使用真实生产环境的变更故障数据集和模拟变更数据集进行了验证. 实验结果表明, 本文方法能够分别以 80% 和 84% 的 Top-3 命中率识别出根因变更, 并且根因变更识别效果显著优于当前最先进的缺陷变更识别方法 (第 5 节).

本文第 2 节介绍软件变更故障相关的实证研究、软件变更前识别缺陷变更、软件变更后定位缺陷变更、服务级别根因定位等相关研究工作. 第 3 节介绍基于大规模即时通信系统微信系统中的真实变更故障实证研究, 得出 5 个关键发现. 第 4 节详细阐述本文提出的在线系统根因变更识别方法. 第 5 节通过实验证明本文提出方法的有效性. 第 6 节针对性讨论根因变更识别真实案例和本文的局限性. 最后, 在第 7 节对全文进行总结, 并简要介绍下一步工作.

2 研究背景和相关工作

在本节中, 本文首先介绍软件变更与软件变更故障的相关背景. 接着, 为了更好地展示本文与已有工作的区别, 将从软件变更实证研究、软件变更前缺陷变更检测、软件变更后缺陷变更检测、服务级根因定位与根因变更识别差异这 4 个方面介绍相关工作.

2.1 软件变更故障

软件变更是指对软件系统进行修改、更新或调整的过程[1]. 这些变更可以涉及软件的功能、性能、安全性、用户界面、数据库结构、代码优化等方面. 软件变更的目的是改进软件系统, 使其更加稳定、可靠、安全, 并且能够满足用户的需求. 软件变更可以分为以下几种类型.

● 模型变更 (model change): 这种变更类型通常涉及修改现有的机器学习模型以提升其性能或准确性. 然而, 引入有缺陷的模型可能导致意外行为, 例如公众号推荐模型不准确导致用户点击率下降.

● 客户端变更 (client change): 这种变更类型涉及更新用户设备上的 Android 或 iOS 应用程序. 有缺陷的客户端变更可能导致兼容性问题或应用程序崩溃等问题, 从而影响用户的使用体验.

● 资源变更 (resource change): 这种变更类型涉及根据运维工程师或自动伸缩方法确定的资源分配的更改. 不适当的伸缩操作可能导致服务过载和请求超时, 从而影响系统的可用性和性能.

● 配置变更 (configuration change): 这种变更类型指的是对服务的配置设置进行的调整. 不正确或不兼容的配置变更可能导致系统不稳定、请求错误或其他意外行为.

● 后端变更 (backend change): 这种变更类型涉及对后端服务进行的修改. 有缺陷的后端变更可能导致服务崩溃或响应时间变慢等问题, 从而影响用户的访问体验.

尽管在软件变更上线之前工程师会对变更的服务进行严格和周密的测试, 但由于生产环境和测试环境存在多种差异和更加复杂的组件交互关系, 有部分缺陷变更仍会通过测试, 并被部署到生产系统中, 然后导致系统出现故障. 本文将由缺陷软件变更引发的故障称为变更故障. 这种故障类型并非由外部因素 (如资源竞争、网络中断等) 导致, 通常是通过执行代码回滚或者上线新的软件版本来进行修复.

2.2 软件变更实证研究

软件变更及其引发的故障一直以来都是软件工程研究的热点, 已有大量高质量的研究工作专注于分析不同场景下软件变更的特征及其引发故障的根因[1,13–15].

Savor 等人[1]介绍了 Meta 和 OANDA 两家公司实施持续部署 (continuous deployment) 的详细经验. 文中通过具体案例全面阐述了两家公司的部署流程、测试策略与发布策略, 并从组织与技术层面深入分析了实施持续部署面临的挑战, 得出实施持续部署可以显著提高部署效率、缩短交付时间、改进产品质量与用户满意度的结论. Li 等人[15]收集并分析了在 Google、AWS 和 Azure 这 3 大云平台上的 354 份公开故障报告, 总结了故障出现的主要原因并进行实证研究. 该论文研究发现 46.6% 的云平台故障涉及软件变更, 这个结果与本文的结果接近. 但是 Li 等人的研究并不是专门针对变更故障的, 没有针对变更故障的具体原因和定位时长等方面进行深入研究.

Zhang 等人[13]通过对 8 个广泛使用的分布式大数据处理系统中 123 个真实世界的变更故障进行了深入研究, 揭示了数据处理系统变更故障的严重性、根因、暴露条件和修复策略. 论文作者观察到, 只有 37% 的软件变更故障在相应版本发布给公众之前被发现, 而大部分 (63%) 变更故障在生产环境中才暴露出来. 并且大部分变更故障是由于数据类型、语义的不兼容 (如序列化库或枚举量定义的改变) 导致的. 与本文的工作相比, Zhang 等人的实证研究主要针对大数据处理系统, 没有考虑为用户提供服务的在线系统, 而这类在线系统也是广泛使用的软件系统, 变更频繁, 急需针对在线系统的软件变更故障进行深入分析.

2.3 软件变更前缺陷变更检测

在软件变更实施前识别潜在缺陷, 以降低变更风险一直是研究热点. 现有相关工作主要从软件测试[16,17]、可靠性审计[8,10,18]和系统验证[13,19,20]等方面来检测缺陷变更.

Rex[19]通过结合机器学习和程序分析的方法, 挖掘故障和错误配置之间的变更规则. Rex 通过两个步骤学习变化规则: 发现变更规则和重新定义变更规则. 在发现变更规则中, 它利用关联规则挖掘技术寻找“频繁”一起更改的文件集. 在发现变更规则后, Rex 执行第 2 步, 即变更规则的重新定义. 它能够将每个粗粒度的、文件级别的变更规则进一步细化, 提高精确性. Rex 分析每个文件中的变化, 确定哪些类型的变化是相关的, 并根据学到的规则向工程师提出建议. 另一方面, PCHECK[10]分析源代码并自动生成配置检查代码 (称为检查器), 以检测潜在的配置错误. 这类软件配置错误检测工具侧重于配置逻辑, 而不是由常见依赖项引起的故障.

Zhang 等人[13]提出了变更前数据缺陷检测工具 DUPChecker 和 DUPTester. DUPChecker 是一种静态分析工具, 用于检查版本升级引发的枚举量改变是否会影响跨版本的数据迁移. 而 DUPTester 会在版本升级之前运行一个工作负载, 生成数据, 并在升级后加载生成的数据, 以检查是否会引发故障. 然而, DUPChecker 和 DUPTester 主要关注软件更新时数据格式的兼容性问题, 并不能对通用的软件变更进行检测. 此外, Batta 等人[8]的研究通过提取变更与引发变更事件之间的联系, 主动评估与软件变更相关的风险.

然而, 由于生产环境与开发环境之间存在的差异 (比如生产环境与开发环境的 CPU 型号不同或操作系统内核版本不同等), 在软件变更前识别缺陷变更的方法无法识别全部的缺陷变更. 除了进行测试和静态分析, 还需要对生产环境的软件变更过程动态地进行监控和检测, 才能及时发现在生产环境才暴露出来的缺陷变更.

2.4 软件变更后缺陷变更检测

对于大规模系统, 部署一个与生产环境具有相同规模和配置的测试环境是困难的. 因此, 当前测试环境和生产环境之间会存在部分差异, 这导致部分缺陷软件变更在测试过程中难以被检测出来, 在部署到生产环境后才被暴露出来. 为了检测在生产环境中暴露的缺陷变更, 目前有一些缺陷变更检测的研究工作专注于识别在软件新版本上线之后出现异常的缺陷变更[9−12,21,22].

FUNNEL[9]采用了 SST (singular spectrum transform) 算法, 用于监测在软件变更后服务的指标是否发生变化. 当检测到异常指标时, 它使用 DiD (difference-in-differences) 算法[23]来比较已变更的服务实例和未变更的服务实例之间的差异, 以判断是否是由软件变更引起的指标变化. 如果是, 则判定该变更存在缺陷, 并通知运维工程师执行回滚操作.

另一方面, SCWarn[12]利用变更服务的多源可观测数据 (如日志和指标) 作为输入, 通过基于历史正常数据训练的 LSTM (long short-term memory) 模型, 挖掘多源可观测数据中的时间依赖性和模式. 在程序变更时, SCWarn 利用训练好的 LSTM 模型来检测变更后的可观测性数据是否异常. 如果异常被发现, 则认为该异常是由变更引起的, 并向运维工程师发出警报. Gandalf[11]持续监测和提取各种故障信号 (如操作系统事件和服务日志), 并使用 Holt-Winters 预测算法基于历史数据来检测发生异常的实例. 如果在服务变更后出现大量该服务的错误实例, Gandalf 会将异常与该服务的变更关联起来.

然而, 现有的缺陷变更检测方法主要将软件变更后的缺陷变更问题视为异常检测任务, 并应用异常检测算法来解决该问题. 缺陷变更检测方法主要负责诊断单次变更是否为缺陷变更. 但这些缺陷变更检测方法通常只考虑单个服务的单个变更是否异常, 而不考虑系统中其他服务变更的传播影响. 而故障传播在相互依赖的在线服务中

经常出现[4], 因此忽视缺陷变更的传播会影响缺陷变更检测算法的诊断, 容易导致出现误报, 进而影响方法的效果. 同时, 缺陷程序在变更完成后才表现出异常, 或者其指标在变更服务中没有表现异常, 现有的缺陷变更检测方法就无法成功识别出缺陷变更. 因此, 仍然需要进一步改进和发展更加全面的方法来解决这些问题.

2.5 服务级别根因定位与根因变更识别差异

在第 2.3、2.4 节中, 我们主要介绍了针对单个变更进行诊断的缺陷变更检测的内容, 在本节中, 本文综述现有的服务级别根因定位方法的研究进展, 并分析其与根因变更识别工作的核心差异. 如图 1(a) 所示, 现有的服务级别根因定位方法通常利用系统的运行时信息 (例如指标和调用链) 作为输入, 但未充分利用系统中存在的变更数据 (如灰度变更的流程和变更时间). 典型的服务级别根因定位方法包括几类: (1) 基于因果推断的根因定位方法[24–27]通过构建指标间的依赖关系来形成因果关系图, 然后计算异常指标间的相关性或采用随机游走算法在图上推断根因服务或服务实例. (2) 基于多维下钻的根因定位方法[28–31]将多维根因分析问题分解为多个单维问题, 通过计算不同指标和属性的预测值与实际值的差异, 以识别最可能导致故障的属性组合进行服务实例级别的根因诊断. (3) 在工业实践中, GMTA[32]和 Canopy[33]是两个基于云原生应用的调用链建立的, 支持工业级云原生应用的流式数据处理、灵活数据访问和高效数据存储的平台. GMTA 将调用链抽象为不同的 Path, 并进一步将其归类为业务流, 可以帮助运维工程师理解系统架构、缩小性能问题的根因范围、检索并可视化故障传播链. 但这项研究的重点是服务级别的根因定位, 并且只使用了异常链路提供的上下文线索. (4) 文献 [4,6,7] 则是通过比较同一种请求的正常和故障调用链的差异来定位服务级别根因.

现有服务级别根因定位方法的局限性在于诊断粒度不足: 其输出多为服务或实例级根因. 在遇到变更故障时, 运维人员需手动关联故障服务与变更之间的因果关系, 导致回滚操作延迟. 本文提出的方法突破该限制, 直接实现变更级诊断, 通过推荐可疑根因变更, 使工程师可快速精准回滚, 避免缺陷版本对系统可用性的持续影响.

3 实证研究

为了深入了解缺陷变更的影响和行为, 本节基于大规模即时通信系统微信的真实变更故障进行实证研究, 得出 5 个关键发现.

3.1 研究问题

本文通过对微信系统内的真实变更故障进行多角度的分析, 旨在回答以下 5 个研究问题.

- 问题 1: 变更故障占总故障数的比重是多少?
- 问题 2: 变更故障是否比其他类型故障更加严重?
- 问题 3: 变更故障是否需要更长的时间来确定其根因?
- 问题 4: 典型的变更故障及其异常表现有哪些?
- 问题 5: 当前的告警系统能否有效定位变更故障?

3.2 数据采集

本文从微信系统内部收集了 2022 年 10 月 1 日–2023 年 3 月 30 日半年期间所有的已经被修复的变更故障. 由于企业数据隐私要求, 本文不披露故障的绝对数量, 而采用相对比例进行量化分析, 以确保数据安全性的同时保持研究的可解释性. 微信是一个为全球数十亿用户提供服务的在线系统, 它包括即时通信、在线支付、短视频、社交圈等多种用户场景. 微信的后端采用微服务架构, 包含了超过 20000 个微服务. 由于企业隐私政策的限制, 本文隐藏了所收集的故障和变更的具体数量等细节.

微信系统为了便于故障分析并防止其再次发生, 当故障发生后, 负责该故障的工程师被要求将故障处理的整个过程以事后总结的形式形成文本记录. 这些事后总结包含故障发生时的关键信息, 例如故障表现、告警时间、告警服务、由负责该故障的工程师确认的最终根因结果、故障严重等级以及其他相关数据. 关于故障分析和变更类型标记, 3 位工程师通过分析事后总结中记录的故障处理步骤和故障原因描述, 在“Cohen’s Kappa 系数”[34]的监

督下, 独立地标记一个故障是否由变更引起以及其变更类型 (问题 1). 对于存在分歧的情况, 本文作者会与负责故障处理的工程师进行讨论, 最终达成共识. 此外, 有一组运维专家根据故障的影响和造成的经济损失确定故障的严重等级 (问题 2). 每个故障的故障识别时间 (问题 3) 可以通过从最终故障检测时间减去告警时间来计算. 告警服务和根因变更服务 (问题 5) 可以分别从事后总结和由负责该故障的工程师确认的最终根因定位结果中提取. 为了获取它们之间的关系, 本文分析了告警服务和根因变更服务之间的调用图.

3.3 问题 1: 变更故障的故障比例

为了回答问题 1, 本文对由缺陷变更引起的故障数量进行量化, 并对这些变更进行详细的分类. 通过分析在 2022 年 10 月 1 日–2023 年 3 月 30 日这 6 个月的时间段内的数据, 本文获得了有关变更故障的分布情况. 图 2 直观地展示了每个月的故障分布情况. 观察结果显示, 在微信系统中, 软件变更是导致故障的一个重要因素, 其比例在 52%–84% 之间变化, 平均为 66%. 换句话说, 相当大比例的故障是由缺陷变更引起的, 这给运维工程师在保障微信的可靠性和性能方面带来相当大的挑战.

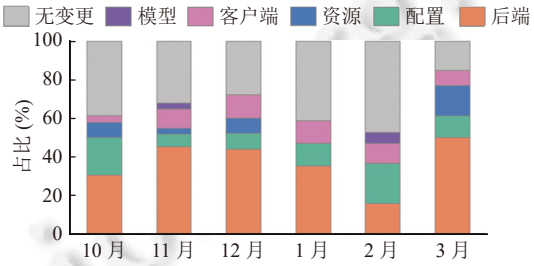

图 2 微信系统中变更故障的分布

通过对不同类型变更的比例进行观察, 本文发现缺陷后端变更占据了很大的比例, 每个月的比例从 15.78% 到 50% 不等. 其次是配置变更, 平均占据 13% 的故障比例. 这是因为这两种类型的变更在软件升级中较为常见, 这种类型的变更次数更多, 因此出现故障的次数也相应变多.

综上所述, 本文可以得出结论: 软件变更是导致故障的重要原因之一. 这使得运维工程师在故障发生时需要优先考虑是否由缺陷变更引起. 因此, 开发一种精准识别缺陷变更的自动化方法至关重要, 这也是本文研究的主要动机.

发现 1. 在微信系统内部, 平均有 66% 的故障是变更故障. 在某一个月内, 高达 80% 的故障是由缺陷变更引起的.

3.4 问题 2: 缺陷更改引发事件的严重程度

在故障发生后, 微信内部有一组运维专家根据故障的影响和造成的经济损失评定故障的严重等级. 表 1 展示了微信系统中的故障严重程度级别的对比情况. 在微信系统中, 故障的严重程度级别分为 1–6 级, 其中 1 级故障代表最严重的故障, 这一级别故障对用户数据安全、公司基础设施或核心业务等造成了更严重的影响.

表 1 由缺陷变更导致的和其他原因导致的故障的严重等级对比 (%)

故障级别	6	5	4	3	2	1
变更故障	52.08	10.41	4.16	0.69	0	0
其他故障	27.77	4.16	0.69	0	0	0

通过观察表 1, 得出以下结论: 与其他故障类型相比, 缺陷变更导致严重故障的比例更高. 具体而言, 6 级变更故障的数量比其他原因导致的故障高出 2.1 倍. 这说明由缺陷变更引起的故障对微信系统的稳定性和可靠性造成更大的威胁. 此外, 5 级和 4 级故障也更多地由缺陷变更引起, 分别比其他原因高出 2.5 倍和 6 倍. 这意味着对于这些故障等级严重的故障, 缺陷变更是主要的根源之一. 值得注意的是, 6 个月内唯一的 3 级故障也是由缺陷变更引起的. 这进一步强调了由缺陷变更引起的故障的严重性和紧迫性. 在处理故障时, 运维团队需要优先考虑和解决由

缺陷变更引起的故障, 因为它们对微信系统的稳定运行具有重大影响.

发现 2. 与其他故障类型相比, 缺陷变更导致严重故障的比例更高.

3.5 问题 3: 变更故障的根因识别时间

为了回答问题 3, 本文需要明确“根因识别时间 (time to identify, TTI)”的定义. 根因识别时间是指从故障发生到运维工程师定位到根因 (如网络中断或模型更改) 所花费的时间. 在图 3 中, 本文展示了由缺陷变更导致的故障和由其他原因导致的故障的根因识别时间的累积分布函数 (cumulative distribution function, CDF) 图.

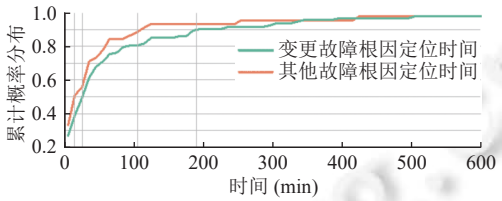

图 3 变更故障和其他故障根因定位时间对比

通过观察图 3, 得出以下结论: 对于大多数故障, 运维工程师往往很难及时确定故障的根因. 具体而言, 对于由缺陷变更导致的故障和由其他原因导致的故障, 至少有 50% 的情况下, 运维工程师需要分别花费 25 min 和 15 min 来确定根因. 而对于 10% 的故障, 这个时间甚至超过了 185 min 和 105 min. 由图 3 可见, 与其他故障相比, 运维工程师需要更长的时间来确定由缺陷变更导致的故障的根因. 这种情况部分归因于当前的根因定位方法在识别由缺陷变更引起的根因方面的不足. 另一个重要因素是一些有缺陷变更的服务不会表现出异常, 而是其上游或下游服务表现异常, 这使得运维工程师更难以确定根因变更.

值得注意的是, 在对这些持续时间较长的定位过程作进一步分析时, 我们发现存在少量需要处理多个变更的故障案例 (总计约 1.8%). 这些多变更故障可以进一步分为两类: 一类是协同变更故障 (约 0.7%), 即由于服务间接口变更不同步或版本不匹配, 需要回滚多个相关服务的变更才能恢复系统; 另一类是级联变更故障 (约 1.1%), 通常发生在一个服务的变更触发了其他服务的自动响应性变更 (如负载过高导致的自动扩容或配置自动调整), 这些连锁反应共同导致了系统故障. 多变更故障比例较低主要得益于微信的工程实践. 首先, 微信采用严格的微服务架构设计, 通过明确的服务边界和标准化的接口契约有效降低了服务间耦合; 其次, 变更管理流程规定高风险变更必须单独发布, 避免了复杂变更的同时上线; 最后, 采用严格的灰度发布策略, 通过分批次部署进一步降低了多变更同时故障的可能性.

总体而言, 这些结果强调了在处理由缺陷变更引起的故障时, 根因变更识别的复杂性和耗时性. 为了加快根因识别的速度, 运维工程师需要在现有服务级别根因定位方法基础上进一步改进和完善, 特别是当故障背后隐藏了多次变更的相互作用时, 更需要结合变更依赖关系、服务依赖关系、变更时序等多种信息来提升故障诊断效率.

发现 3. 定位由缺陷变更导致的故障的根因需要比由其他原因导致的故障花费更长的时间.

3.6 问题 4: 典型软件变更故障及其表现

为了进一步深入理解变更故障的表现, 本文系统地总结了一系列典型变更故障, 详见表 2 中的前 9 行. 通过这个表, 可以观察到系统的业务指标 (如请求响应成功率或请求延迟) 以及机器指标 (如 CPU 利用率或内存利用率) 都可能会受到软件变更的影响.

在某些故障情况下, 已变更的服务实例与未变更的服务实例在软件灰度变更阶段会展现出明显的指标差异, 这种差异可以用来推断变更是否出现异常. 以后台变更引入错误的参数顺序为例, 已变更的服务实例会出现函数调用失败和请求响应成功率下降的表现, 而未变更的服务实例则保持稳定的请求响应成功率. 通过对比已变更与未变更的服务实例的请求响应成功率差异, 运维工程师可以推断出该变更是缺陷变更.

此外, 还存在部分变更故障在灰度变更阶段不表现异常, 而是在变更完成后才显现出问题. 举例来说, 当程序

员在新增的代码中引入导致内存泄露的代码缺陷时, 内存泄露会导致服务的内存使用率增加, 请求响应时间延长, 最终可能因为内存耗尽而导致请求响应成功率下降. 然而, 由于内存泄露需要一段时间才能显现出明显异常, 这种异常可能会在程序变更完成后才出现. 因此, 在程序变更完成后, 持续对服务进行监控是必要的.

表 2 典型软件变更故障及其表现

变更类型		变更内容	异常表现
缺陷变更	后台变更	遗漏函数参数	函数调用失败, 请求响应成功率下降
		错误的参数顺序	函数调用失败, 请求响应成功率下降
		新代码内存泄露	内存使用率缓慢上升, 请求延迟升高, 请求响应成功率下降
		遗漏处理异常	变更服务频繁崩溃, 服务请求失败, 请求响应成功率下降
	配置变更	错误的调用端口	服务请求失败, 服务连接失败, 请求响应成功率下降
		错误的配置文件	读取配置文件失败, 服务启动失败, 请求响应成功率下降
		错误的访问密钥	请求下游服务失败, 请求响应成功率下降
	资源变更	减少服务实例数目	变更服务资源使用率上升, 请求处理延迟升高
	模型变更	上线错误推荐模型	用户点击率下降, 广告收益下降
正常变更	后台变更	新增调用逻辑	响应成功率保持稳定, 请求响应延迟升高
	资源变更	更换高性能服务器	请求响应成功率保持稳定, CPU利用率下降, 请求响应延迟降低

另一个观察结果是, 变更服务在指标表现上的异常并不一定意味着这次软件更改存在缺陷. 这是因为一些软件更改可能会导致监控数据异常, 但这种异常是符合预期的行为. 例如, 用新的高性能服务器替换旧服务器可能会导致 CPU 使用率下降和响应延迟下降, 但这种下降是符合预期的. 表 2 的最后 2 行列举了两个这种类型的软件变更. 因此, 运维工程师应该考虑如何过滤掉预期的变化, 以减少误报的发生.

发现 4. 在指标上表现异常的变更服务并不一定意味着这次的软件变更存在缺陷.

3.7 问题 5: 告警系统识别缺陷变更的效果

为了回答问题 5, 首先对微信中的告警系统进行简单介绍. 微信内部告警系统采用结合有监督的图神经网络模型和灵活的转移矩阵设计的 PageRank 算法来实现根因定位. He 等人[35]在其相关论文中对微信告警系统进行了详细介绍. 在问题 5 中, 本文对微信告警系统输出的根因服务与根因变更对应服务之间的差异性进行了分析. 需要注意的是, 在问题 5 中, 本文排除了由客户端更改导致的故障, 因为微信的告警系统主要负责监控和管理微信后台服务. 此外, 不会触发告警的缺陷变更也被排除在分析范围之外.

图 4 展示了微信告警系统输出结果与根因变更对应服务之间的差异性. 在图中, 标签“告警服务”表示告警服务与根因变更服务相同. 而“上游 i”表示根因变更服务是距离告警服务 i 层的上游服务. 以一个例子来说明, 假设服务 A 调用 B, 而 B 又调用 C, 如果告警服务是 C, 而根因变更服务是 A, 那么可以将 A 表示为“上游 2”. “下游 i”表示根因变更服务是距离告警服务 i 层的下游服务. “其他”表示根因变更服务位于告警服务的两层调用之外或没有调用关系.

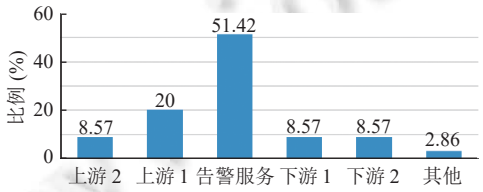

图 4 根因变更服务与告警服务的位置关系

图 4 的结果显示, 对于约 51% 的事件, 告警系统能够将根因变更对应的服务识别出来, 但并不能识别是否与缺陷变更相关. 对于其他情况, 约 47% 的根因变更服务位于告警服务的两层调用之内的上游或下游服务中. 此外, 大约有 2% 的根因变更服务位于告警服务的两层调用之外, 或者与告警服务没有任何调用关系. 这一发现揭示了

在微信告警系统中, 绝大多数根因变更服务与告警服务之间存在着较近的调用关系. 然而, 需要进一步的分析来确定这些服务之间的因果关系. 这对于改进告警系统的根因定位能力具有重要的指导意义.

发现 5. 绝大多数根因变更对应的服务与告警服务之间存在较近的调用关系.

4 根因变更识别方法

受第 3 节实证研究中 5 个发现的启发, 当前工业界迫切需要一种专门针对变更故障的根因变更识别方法, 以便快速定位导致变更故障的根因. 本节以服务依赖图和服务变更信息为输入, 提出了一种智能的根因变更识别算法, 旨在帮助运维工程师快速和准确地定位导致变更故障的根因变更.

4.1 问题定义

在介绍根因变更识别框架前, 本文首先对两个关键术语进行区分: 缺陷变更是指任何在功能或行为上引入错误的软件修改. 根因变更则特指在系统发生故障后, 通过综合分析系统运行时信息和变更上下文, 最终被确定为引起该特定故障的根本原因的缺陷变更. 接着, 本文对根因变更识别问题进行定义, 对一个在线系统包含 M 个服务的集合 S, 给定包含 N 个变更的集合 C, 以及该系统的服务依赖图 G 和 KPI (key performance indicator) 数据 K, 那么关于一个变更故障的根因变更识别问题可以定义为:

$$\mathcal{F}:(C,S,G,K)\rightarrow Score \tag{1}$$

其中, $Score=\{(Service,Change_i,Score_i)\}_{i=1}^{N}$ 是一个根因变更的得分列表. 列表中的实例不仅包含了根因服务, 还明确了服务对应的可疑变更, $Change_i$ 分别表示第 i 个变更, $Service$ 表示 $Change_i$ 关联的服务, $Score_i$ 表示 $Change_i$ 变更的可疑得分, 得分越高意味着这个变更越有可能是根因变更. 这个得分列表能够为运维工程师提供故障诊断参考, 优先检查可疑得分较高的变更, 这能够加快根因变更的回滚, 加快故障修复速度, 缩短用户受故障影响的时长.

4.2 研究挑战

在第 4.1 节中, 本文对根因变更识别的问题进行了定义. 但是想要在大规模在线系统中解决这个问题主要还存在以下几个挑战.

● 挑战 1: 系统规模大、服务间依赖复杂、系统中同时存在多个变更. 在大规模在线系统中, 通常涉及大量的服务和组件, 这些组件之间存在复杂的依赖关系, 这增加了根因变更识别的挑战, 因为一个变更可能会对多个服务产生影响, 并且可能会引发级联故障. 除此之外, 在大规模系统中通常在一段时间内同时存在多个变更, 在这种情况下, 确定哪个变更是导致问题的根因变得复杂.

● 挑战 2: 根因变更不表现出 KPI 异常. 在问题 3 中, 本文发现部分存在根因变更的服务并不表现出 KPI 异常, 而是其上游或下游服务表现异常. 这意味着仅考虑 KPI 将无法发现这种类型的根因, 需要挖掘其他数据源提供的线索来定位这种类型的根因.

● 挑战 3: 表现出 KPI 异常的变更并非是根因变更. 在一个复杂的系统中, 一个服务在变更后该服务 KPI 出现异常除了由缺陷变更导致之外, 还有其他外部因素 (例如故障传播或资源竞争) 也会导致 KPI 异常. 因此无法通过简单判断 KPI 是否异常来判定当前的变更就是根因变更.

当前故障根因分析方法[4–7]虽然能够有效地解决挑战 1 中复杂依赖关系带来的挑战, 但是由于没有考虑变更数据, 直接将故障关联到根因变更. 而缺陷变更检测方法[8–10,12]虽然能够在一定程度上解决挑战 3, 但是由于其没有考虑到服务间的依赖关系, 对挑战 2 束手无策. 因此, 需要更先进的变更故障诊断框架来解决以上的挑战, 帮助运维工程师快速回滚根因变更, 提升系统的可用性.

4.3 方法概述

为了解决已有方法的不足和上述的挑战, 本文同时考虑服务依赖图信息和变更信息, 提出了一种自动化根因变更识别方法. 图 5 展示了本文提出的根因变更识别方法的总体框架. 这个根因变更识别方法由当前成熟的告警

算法 GIED[35]触发. GIED 是一个使用有监督的图神经网络模型进行异常检测, 并结合灵活的转移矩阵设计的 PageRank 算法进行服务级别根因定位的算法. 本文之所以沿用一个成熟的告警算法而非从零开始设计, 是因为根据发现 5 的结果, 当前的成熟告警算法能够有效约简搜索空间. 根据发现 5 的结果, 在获得 GIED 算法输出的告警服务后, 根因变更识别算法只需考虑告警服务及其上下游两层调用的服务的变更, 从而能够有效地缩小搜索空间, 加快搜索速度.

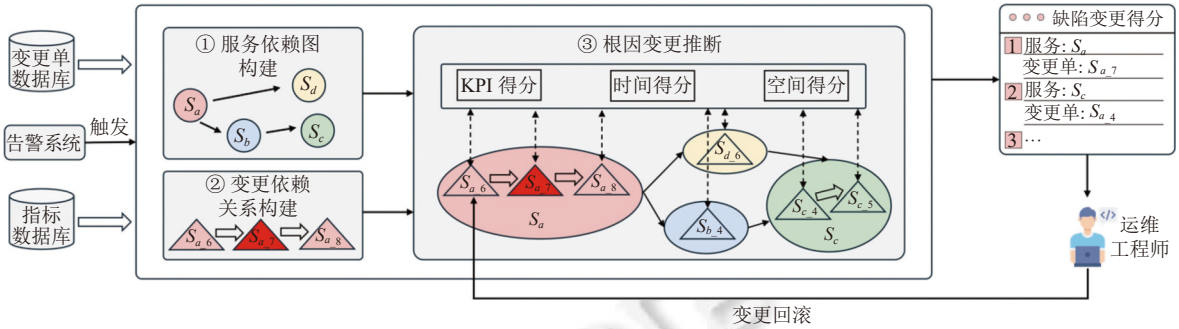

图 5 根因变更识别方法总体框架

本文提出的根因变更识别框架主要包括服务依赖图构建模块, 变更依赖构建模块, 根因变更推断模块 3 个模块. 在被告警系统触发后, ① 服务依赖图构建模块会以告警服务为中心构建上下游的服务依赖图 (第 4.4 节). ② 针对服务依赖图上的所有服务, 如果服务在 H h 内有变更, 变更依赖构建模块会构建变更之间的依赖关系 (第 4.5 节). ③ 根因变更推断模块则会对每个服务的 H h 内的所有变更计算 KPI 差异得分, 时间衰减得分和空间依赖得分. 然后将这 3 个得分输入根因变更推断模块, 输出一个根因变更得分列表供运维工程师检查 (第 4.6 节). 在理想情况下, 根因变更得分排名第 1 的变更就是导致这次故障的根因变更. 运维工程师可通过回滚或快速修复根因变更使系统恢复正常, 避免对用户的持续影响, 提升系统的可用性.

4.4 服务依赖图构建

在告警系统被触发后, 告警系统将输出一个最可疑的服务让运维工程师检查. 本文将这个告警系统输出的服务称为“告警服务”. 本节的动机来自对历史故障数据的深入分析 (发现 5): 98% 的根因变更都集中在告警服务的直接上下游两层调用范围内. 这一发现具有重要的工程实践价值, 它表明在故障诊断时, 只需要重点关注告警服务周边的局部调用拓扑, 就能以较高概率定位到问题根源. 基于这一发现, 本文的服务依赖图构建模块采用了一种聚焦式的构建策略: 以告警服务为中心节点, 向外扩展构建两层深度的服务依赖图. 这种局部化的构建方法相比全系统范围的依赖分析, 可以显著降低计算复杂度, 同时保持较高的根因变更检出率.

在具体构建服务依赖图时, 我们的构建模块采用多阶段处理流程: 首先, 模块会从分布式追踪系统 (如 Jaeger) 和服务网格控制平面 (如 Istio Pilot) 中抽取服务间的实时调用关系数据. 在现代云原生环境中, 这些调用关系可以通过多种技术手段可靠获取: (1) 对于基于 Spring Cloud 等微服务框架的应用, 可以通过解析 REST API 调用日志或 Feign 客户端代理信息来建立调用关系; (2) 对于服务网格管理的服务, 可以直接从 Istio 或 Linkerd 的控制平面获取服务间的流量路由规则; (3) 通过分布式追踪系统 (如 OpenTelemetry、SkyWalking) 收集的调用链 (Trace) 数据, 可以精确还原服务间的调用拓扑和时间依赖关系. 在构建出完整的二层调用关系图后, 模块会进一步查询变更管理系统 (如企业内部 CMDB 或 GitOps 工具链), 获取该调用图上所有服务在最近 H h 内的变更记录. 基于对运维团队的调研访谈, 我们将 H 的默认值设定为 48, 这个时间阈值得到了运维经验的强有力支持: 真实故障统计显示约 95% 的变更相关故障都会在变更后的 48 h 内显现出来. 对于调用关系图中的叶节点 (即只被调用但不调用其他服务的末端节点), 如果它们在过去 H h 内没有关联任何变更操作, 构建模块会执行剪枝优化: 递归地将这些无变更的叶节点从图中移除. 通过这种剪枝处理, 可以显著减少后续计算需要处理的节点数量 (平均减少 62% 的节点), 从而大幅提升整体分析效率. 最终, 构建模块会将剪枝后的调用关系图与变更信息进行整合, 生成一个服务

依赖图, 其中节点代表服务, 边代表调用关系. 这个精炼后的依赖图将作为核心输入, 传递给后续的变更评分模块进行更精细化的根因分析.

4.5 变更依赖关系构建

由于服务功能的快速迭代, 现代微服务架构中的单个服务可能会在极短时间窗口内 (如 48 h) 经历多次变更部署. 这种高频变更特性使得故障诊断面临新的挑战: 当系统出现异常时, 运维团队不仅需要定位故障服务, 还需要在众多变更中准确识别导致问题的具体变更版本. 在获得完整的服务依赖图后, 我们可以通过集成变更管理系统 (如 GitOps 工具链或企业内部的变更单系统) 查询服务依赖图上所有服务在最近 H h 内发生的变更列表. 值得注意的是, 一个服务的多次变更之间往往不是相互孤立的, 而是存在复杂的版本依赖关系链. 例如, 假设服务 A 的变更 1 在 24 h 前开始部署, 并在 12 h 前完成; 随后, 服务 A 的变更 2 在变更 1 完成后立即开始部署. 这种情况下, 变更 2 在版本上直接依赖于变更 1 的基础代码状态. 如果最终发现是变更 1 引入了缺陷, 那么仅回滚到变更 2 之前的版本并不能彻底解决问题, 因为变更 2 是基于已经存在缺陷的变更 1 版本进行的迭代. 此时, 必须回滚到变更 1 之前的原始稳定版本, 系统才能完全恢复正常. 因此, 在分析故障时, 对同一个服务的多个变更, 必须首先通过代码仓库的提交历史、构建流水线的依赖关系或部署系统的版本记录, 精确还原它们之间的版本依赖图谱.

为了获得同一个服务的多个版本之间的依赖关系, 需要建立系统化的版本溯源机制. 具体而言, 可以从以下 3 个维度进行依赖关系提取: (1) 代码仓库维度: 通过分析 Git 提交历史中的分支合并记录和 tag 标记, 确定各版本间的代码继承关系; (2) 构建流水线维度: 检查 CI/CD 系统中各次构建的输入输出依赖, 特别是容器镜像的层级依赖关系; (3) 部署配置维度: 比对 Kubernetes 等编排系统中的 Deployment 版本更迭记录, 识别滚动更新过程中的版本替换顺序. 这 3 个维度的数据可以通过调用各系统的开放 API 进行自动化采集, 并存储为有向无环图 (DAG) 的形式, 其中节点代表特定版本, 边代表版本间的依赖关系. 例如, 当服务 A 的 v1.2 版本是基于 v1.1 版本的代码构建, 而 v1.3 又依赖于 v1.2 时, 就会形成 v1.1→v1.2→v1.3 的依赖链. 这种显式的版本依赖图谱不仅可以帮助快速定位需要回滚的目标版本, 还能在变更前进行影响评估, 预测某个变更可能波及的依赖服务范围. 在实际实现中, 可以开发专门的版本依赖分析器, 定期从各子系统同步元数据, 并建立统一的版本依赖知识图谱, 为故障诊断提供更全面的上下文信息.

4.6 根因变更推断

在完成服务依赖图的构建和变更依赖关系的分析后, 系统已经具备了完整的拓扑结构和变更上下文信息. 接下来的核心挑战是从众多候选变更 (通常每个故障事件涉及上百个时空相关变更) 中准确定位最可能导致故障的根因变更. 本文在根因变更推断中综合考虑了变更前后 KPI 差异、变更时间与故障时间的相关性和服务依赖强度这 3 个关键维度的因素, 这些因素共同构成了一个多维评估框架.

4.6.1 KPI 差异得分计算

为了降低部署新软件版本所带来的风险, 灰度变更已经成为软件部署中一种流行的技术手段. 在灰度变更中, 工程师们首先只在一部分服务实例上部署软件变更, 而不是立即更新整个服务的所有实例. 这样做的好处在于, 如果已变更的服务实例引发异常, 可以及时回滚. 如果没有出现异常, 就可以逐步将变更推广到属于该服务的所有实例, 直到所有服务实例都完成更新.

大多数变更故障通常会在灰度变更过程中显现出来. 其典型表现是已变更服务实例组和未变更服务实例组之间出现明显的性能或可靠性差异. 例如在图 6 中, 服务 A 上线了一个缺陷变更, 当前处于灰度变更过程中. A_1 和 A_2 是已变更服务实例, A_3 和 A_4 是未变更服务实例. 图中的曲线表示每个服务实例的请求延迟. 此次服务 A 的变更引发系统延迟上升, 进而触发告警系统. 通过分析服务 A 中已变更服务实例组和未变更服务实例组之间的请求延迟可以发现, 已变更的 A_1 和 A_2 的请求延迟剧烈上升, 而未变更的 A_3 和 A_4 的请求延迟保持稳定. 通过对比 A_1 和 A_2 与 A_3 和 A_4 可以发现只有已变更的服务实例出现异常, 因而可以推断异常是由 A 的缺陷变更导致的.

为了检测出已变更和未变更服务实例组之间是否出现显著的差异, 本文采用了一种分流思想[36]对比测试组 (已变更服务实例) 和对照组 (未变更服务实例) 的性能表现. 其核心思路是, 在灰度变更过程中, 如果一个故障是由

缺陷软件变更导致的, 那么它只会影响到已变更服务实例的性能, 而不会影响到未变更服务实例的性能. 而如果一个故障不是由缺陷软件变更导致的 (例如网络中断), 那么已变更和未变更的服务实例的性能都会受到该故障的影响.

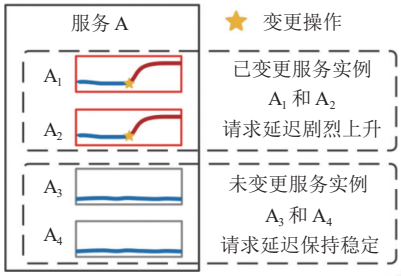

图 6 服务 A 在灰度变更过程中服务实例请求响应延迟曲线的变化

具体来说, 本文采用了 DiD 算法[23], 它是经济学领域中用于评估在特定时间点实施的干预措施效果的常用方法. 在本方法中, DiD 用于比较已变更服务实例组与未变更服务实例组的 KPI 随时间的变化, 并将 KPI 差异归因于软件变更的实施. 对给定同一个服务不同服务实例的 KPI 曲线, DiD 首先根据变更时间和变更操作对它们进行如下划分.

● 测试组和控制组: 测试组由已变更的服务实例组成, 用 $T=1$ 表示. 控制组由尚未变更的服务实例组成, 用 $T=0$ 表示.

● 变更后区间和变更前区间: 变更后区间表示变更操作实施后的时间段, 用 $P=1$ 表示; 变更前区间表示变更操作实施前的时间段, 用 $P=0$ 表示.

令 K_i 为 KPI i 的测量值, 那么 DiD 模型可以表示为:

$$K_i=\alpha_0+\alpha_1 T+\alpha_2 P+\alpha_3 T\times P+\epsilon_i \tag{2}$$

其中, α_0 表示控制组在变更前的基础值. α_1 和 α_2 分别表示变更和时间的主要影响. α_3 是 DiD 影响估计值, 它表示变更对影响的因果效应, 它可以形式化为:

$$\alpha_3=(E[Y|C=1,P=1]-E[Y|C=1,P=0])-(E[Y|C=0,P=1]-E[Y|C=0,P=0]) \tag{3}$$

其中, $E(\cdot)$ 表示期望值. 如果 KPI 的值变化是由软件变更以外的因素造成的, 那么已变更服务实例和未变更服务实例的相对性能不会有明显变化. 此时 α_3 的值应该接近于 0. 相反, 如果 α_3 远远大于 0 或者小于 0, 那么可以判定相对性能的差异是由于变更导致的. 本文设置了一个阈值 λ 来判断 α_3 是否显著, 我们选取 DiD 方法中典型的阈值 $\lambda=0.15$. 如果一个已变更服务实例的 $\alpha_3>\lambda$, 那么就判定这个服务实例为由缺陷变更导致的异常服务实例.

在轮询判断每个已变更的服务实例以后, 可以获得所有的变更异常的已变更服务实例的数目. KPI 差异得分计算模块计算得分的核心思想是: 如果一个服务被判定为由变更导致的异常已变更服务实例越多, 那么这个服务的变更越有可能是根因变更. 具体的 KPI 差异得分计算的逻辑可以形式化为:

$$Score_{\text{diff}}=\frac{N_{\text{apost}}}{N_{\text{post}}} \tag{4}$$

其中, N_{apost} 是被判定为由变更导致的异常的已变更服务实例的数目, N_{post} 是所有已变更服务实例数目. 如果一个服务异常的已变更服务实例数目越多, 它的差异得分越高. 通过计算差异得分, 能够解决挑战 3, 由外部因素和故障传播导致的 KPI 异常服务将获得低的差异得分.

4.6.2 时间衰减得分计算

对于部分隐蔽性较强的变更故障, 其故障模式往往呈现出典型的“间接传播”特征. 这类故障不会立即在变更服务的直接 KPI 指标上显现异常, 而是通过服务依赖链进行潜在传播, 最终在其他上下游服务表现出可见故障. 这种情况下仅考虑 KPI 的线索来进行根因变更识别是不够的. 为了有效捕捉这类隐蔽故障模式, 本节引入时间维度作为关键分析因素. 它的核心思想基于“时间衰减效应”, 变更与故障的时间间隔越短, 关联性越强. 如果一个服务变更后, 系统很快出现异常, 那么该变更与系统异常会有更强的时间关联性. 相反, 如果变更很长时间后系统才出

现异常, 那么可以认为该变更与系统异常的时间关联性不强. 因此, 对给定告警时间 $time_a$, 一个变更的时间得分可以根据该变更开始的时间 $time_c$ 形式化为如下:

$$Score_{\text{time}} = \frac{H - Time(time_a - time_c)}{H} \tag{5}$$

其中, $Time(\cdot)$ 计算了 $time_a$ 和 $time_c$ 相差的小时数. 基于公式 (5), 本方法给发生在告警时间附近的变更赋予更高的可疑变更得分, 给发生在远离告警时间的变更赋予更低的可疑变更得分.

4.6.3 空间依赖得分计算

除 KPI 和时间的因素以外, 本方法还考虑了变更服务在服务依赖图上的空间得分. 在图 4 中可以观察到, 告警服务和根因变更的服务存在“拓扑邻近性”, 也就是距离告警服务越近的服务的变更更有可能是根因变更. 因此本文在定位根因变更时还考虑了空间的因素. 在获得服务依赖图后, 本文可以将空间得分形式化为:

$$Score_{\text{spatial}} = \frac{L}{L + distance} \tag{6}$$

其中, L 表示服务依赖图中距离告警服务最大的跳数, $distance$ 表示变更服务与告警服务的距离. 在第 4.4 节中本文介绍了服务依赖图构建模块会以告警服务为中心构建两层服务依赖图, 因此 L 的默认取值为 2. 如果变更服务与告警服务是同一个服务, 那么 $distance$ 的值为 0. 其他情况, $distance$ 的值为变更服务与告警服务之间跳数的绝对值. 通过计算时间得分和空间得分, 能够让方法考虑 KPI 没有异常的根因变更 (挑战 2), 使这种静默的变更获得更高的可疑得分.

在获得 KPI 差异得分、时间得分和空间得分后, 根因变更推断模块将整合这 3 个得分, 从而获得最终的可疑变更得分列表. 根据不同得分在根因变更识别方法中的权重, 对变更 $Change_i$, 它的根因变更得分形式化为:

$$Score_i = \alpha Score_{\text{diff}} + \beta Score_{\text{time}} + \delta Score_{\text{spatial}} \tag{7}$$

其中, α、β、δ 分别是 KPI 差异得分、时间得分和空间得分的权重. 在微信系统中, 运维工程师经验性地认为在根因变更识别中 KPI 差异得分、时间得分和空间得分对根因的贡献相同, 因此本文将 3 个得分的权重都设置为 1. 在其他系统中, 可以根据系统的特性对权重进行调整. 此外也可使用一些自动化的参数调整方法来调整权重, 受篇幅限制本文不详细讨论这部分内容.

在遍历完服务依赖图上的全部变更后, 根因变更推断模块会对可疑得分进行排序, 得到根因变更得分列表 $Score$. 在理想情况下, 可疑得分最高的服务的变更为根因变更. 算法 1 展示了根因变更识别算法的整体过程.

算法 1. 根因变更识别算法.

输入: 服务调用关系 R, KPI 时序数据 K, 服务变更数据 C, 告警服务 $Service_a$, 告警时间 $Time_a$, DiD 算法阈值 λ;
输出: 可疑变更得分列表 $Score$.

1. G ←服务依赖图构建 (R, C, $Service_a$)
2. **FOR** ($Service$, $Change_i$) **IN** G
3. $Score_{\text{diff}}$ ← KPI 差异得分计算 (K, λ)
4. $Score_{\text{time}}$← 时间衰减得分计算 ($Change_i$, $Time_a$)
5. $Score_{\text{spatial}}$ ← 空间依赖得分计算 (G, $Service$)
6. $Score_i$ ← 根因变更推断 ($Score_{\text{diff}}$, $Score_{\text{time}}$, $Score_{\text{spatial}}$)
7. $Score.Add$(($Service$, $Change_i$, $Score_i$))
8. **ENDFOR**
9. $Score.Sort$()
10. **RETURN** $Score$

获得的可疑得分列表能够指导运维工程师检查对应的服务的变更, 从而加快根因变更识别的速度. 在识别到

根因变更后, 运维工程师还需要考虑第 4.5 节中变更之间的依赖性决定回滚的版本. 如图 5 所示服务 S_a 在故障发生前 48 h 内存在 3 次变更, 根因变更得分列表 *Score* 中可疑得分排名前二的是变更 S_{a_8} 和 S_{a_7}, 运维工程师检查变更后判定 S_{a_7} 是根因变更, 需要回滚 S_{a_7} 到 S_{a_6}. 同时考虑到变更之间的依赖性, 即变更 S_{a_8} 依赖于 S_{a_7}, 也需要将 S_{a_8} 回滚到 S_{a_6}, 这能够加速系统恢复的过程, 减少故障影响范围.

5 实验分析

本节将用从微信系统中采集的真实变更故障数据集测试本文提出的方法, 并回答以下问题.

- 问题 6: 本文提出的方法在根因变更识别上的准确性如何?
- 问题 7: 本文提出的 3 个可疑得分对根因变更识别的贡献如何?
- 问题 8: 本文提出的方法是否容易受参数影响?

5.1 实验设置

- 微信变更故障数据集: 本文从为十几亿用户提供服务的微信系统的真实环境中采集变更故障数据用于验证本文提出的方法. 在微信中, 软件变更频率每天可达数千次. 在微信运维工程师的配合下, 本文采集了包括 30 个带标注的变更故障的真实数据集. 本文数据集中不包含多变更故障, 这是因为根据第 3.5 节的分析, 多变更故障在真实故障中的占比较小 (约 1.8%), 存在一定的偶发性, 本文的数据采集阶段没有遇到多变更故障. 表 3 中展示了本文采集变更故障的详细信息, 包括缺陷变更发生故障的阶段、变更类型和数量的具体分布情况. 这些故障发生在即时聊天、移动支付、短视频、公众号等多个领域, 这使得实验的结果更有代表性和泛化性. 每个变更故障数据集都包含本方法所需的告警服务、变更信息、服务请求成功率等指标数据以及由运维工程师手动标注的根因变更. 对每个故障, 本文还采集了以告警服务为中心两跳范围的所有服务在告警前 48 h 内的所有变更. 平均每个变更故障在时空范围内关联 218 个变更, 手动从这些变更中定位根因变更需要耗费大量的人力.

表 3 微信变更故障数据集中缺陷变更的发生阶段、变更类型和数量分布

故障发生阶段	变更类型	故障数量
灰度变更	模型变更	0
	资源变更	2
	配置变更	2
	后台变更	12
变更完成	模型变更	2
	资源变更	2
	配置变更	2
	后台变更	8

- 模拟变更故障数据集: 本文在一个拥有 10 个节点的 Kubernetes 平台上部署了开源微服务基准测试系统 OnlineBoutique (OB) (https://github.com/GoogleCloudPlatform/microservices-demo). OB 模拟了一个电子商务微服务系统, 用户可以在其中浏览、查看和购买产品. OB 已在许多先前研究中被广泛使用[4,5]. 因 OB 应用不涉及模型变更, 基于先前对真实变更故障的研究[37,38], 本文精心设计并模拟了 OB 应用中不同服务的缺陷后台变更、缺陷配置变更和缺陷资源变更. 此外, 本文还将正常变更纳入数据集以确保其多样性. 对于每个案例, 我们随机选择一个缺陷变更以及 2–5 个不同服务上的正常变更, 形成了一个多样化且全面的数据集. 模拟变更故障数据集共包含 50 个故障案例.
- 评价指标: 本文采用了一种广泛使用的评价指标 Top-*k* 命中率 (*HR@k*) 来评估本方法和对比方法的性能[4]. *HR@k* 指的是根因变更出现在根因列表结果中前 *k* 位的概率. *HR@k* 可以形式化为如下:

$$HR@k = \frac{1}{N}\sum_{i=1}^{N}\left(C_i \in Score^i_{[1:k]}\right),$$

其中, C_i 是第 i 个故障的根因变更, $Score^i_{[1:k]}$ 是第 i 个故障的前 k 个结果列表, N 是所有故障的数量. 本文设置 k=1,

3, 5, 因为根据微信运维工程师团队的调查, 超过 73% 的运维工程师只考虑前 5 个排名的结果. *HR@k* 的结果越高, 方法的效果越好.

● 实验实施: 本文基于 Python 3.6 实现了本文提出的根因变更识别方法的原型系统. 本文所有的实验是在一台腾讯云虚拟机上进行的, 该虚拟机配备有 32 核的 AMD CPU (2.6 GHz) 和 32 GB 运行内存, 1 TB 的硬盘, 运行 Ubuntu 20.04 操作系统.

5.2 对比方法

在实验中, 本文选择以下 3 种不同的对比方法来和本文提出的方法进行对比分析.

(1) FUNNEL[9]: 该方法采用 SST 算法监测在软件变更后服务指标是否发生变化. 当检测到异常指标时使用 DiD 算法来比较已变更的服务实例和未变更的服务实例之间的差异, 以判断是否是由缺陷软件变更引起的指标变化. FUNNEL 在百度内部网络变更中得到充分验证, 代表了工业缺陷变更检测系统的前沿技术.

(2) Gandalf[11]: 该方法持续监测和提取各种故障信号 (如操作系统事件和服务日志), 并使用 Holt-Winters 预测算法基于历史数据来检测发生异常的实例. 如果在服务变更后出现大量该服务的错误实例, Gandalf 会将异常与该服务的变更关联. Gandalf 在 Microsoft Azure 中被大规模实际应用和部署, 代表了工业缺陷变更检测的前沿技术.

(3) SCWarn[12]: 该方法利用变更服务的多源可观测数据 (如日志和指标) 作为输入, 通过基于历史正常数据训练的 LSTM 模型来检测变更后的可观测性数据是否异常从而推断根因. SCWarn 代表了当前基于深度学习方法的缺陷变更检测前沿技术.

5.3 问题 6: 有效性评估

表 4 展示了本文提出的方法和对比方法在微信变更故障数据集上根因变更识别的准确率. 从表 4 中可以看出, 与当前最先进的方法相比, 本文提出的方法在 *HR*@1 上取得了更好的结果 (73.33%). 具体而言, 本文提出的方法在 *HR*@1 方面比 FUNNEL 提升了 29.4%, 比 Gandalf 和 SCWarn 提升了 9.99%. 表 5 展示了模拟数据变更数据集上的根因变更识别结果, 从表中可以看出本文方法同样表现出色. 在 OnlineBoutique 基准系统数据集的 50 个故障案例中, 本文方法达到了 74% 的 *HR*@1 和 84% 的 *HR*@3, 明显优于基线方法, 展现了强大的跨系统泛化能力. 本文方法的优越性源于其对根因变更的定位方式. 之前的方法只关注在指标上有异常变化的变更, 而未考虑到一些缺陷变更通过故障传播在其他服务上表现出异常的情况. 本文方法通过综合考虑服务依赖图和变更时间, 弥补了之前方法的不足, 从而取得更好的根因变更识别效果. 尤其在复杂的服务依赖关系中, 本文提出的时空特征提取和变更可疑度评分机制能够更准确地捕捉故障传播路径, 有效识别出真正的根因变更, 即使该变更本身的异常指标不明显. 实验证明, 这种方法在实际生产环境和模拟环境中都具有显著优势.

表 4 微信变更故障数据集根因变更识别准确率对比结果 (%)

方法	*HR*@1	↑*HR*@1	*HR*@3	↑*HR*@3	*HR*@5	↑*HR*@5
FUNNEL[9]	56.67	**29.40**	73.33	**9.10**	76.67	**13.04**
Gandalf[11]	66.67	9.99	76.67	4.34	80.00	8.33
SCWarn[12]	66.67	9.99	73.33	9.10	80.00	7.33
本文方法	**73.33**	—	**80.00**	—	**86.67**	—

注: ↑表示准确率提升的百分比

表 5 模拟变更故障数据集根因变更识别准确率对比结果 (%)

方法	*HR*@1	↑*HR*@1	*HR*@3	↑*HR*@3	*HR*@5	↑*HR*@5
FUNNEL[9]	60.00	**23.33**	68.00	**23.52**	72.00	**22.22**
Gandalf[11]	64.00	15.62	72.00	16.66	80.00	10.00
SCWarn[12]	62.00	19.35	70.00	20.00	78.00	12.82
本文方法	**74.00**	—	**84.00**	—	**88.00**	—

注: ↑表示准确率提升的百分比

在微信变更故障数据集上的 *HR*@5 方面, 本文提出的方法达到了 86.67% 的命中率, 这意味着对于大部分故障, 运维工程师可以通过检查可疑列表来发现故障. 对于本文方法未能准确定位的变更故障, 本文作者与负责故障处理的工程师进行了深入讨论, 探索未能定位成功的原因. 研究发现, 未能定位成功的原因主要有两项: (1) 缺陷变更在变更完成 48 h 后系统才出现异常, 超出了本方法的关注时间范围. 这类故障可能是由于缺陷变更引入的内存泄漏或资源占用后未及时释放的缺陷代码所导致, 这种缺陷代码不会立即显现, 而是在长时间运行过程中才会暴露出来; (2) 由于本方法依赖的告警系统将根因变更导致的告警视为环境噪声导致的, 对告警进行了过滤, 因此未触发本文提出的根因变更方法.

5.4 问题 7: 消融实验

表 6 中展示了在两个数据集上缺少差异得分计算模块、时间得分计算模块或空间得分计算模块的变种方法的根因变更识别准确率, 从而揭示这 3 个得分对本方法的贡献. 从表 6 中可以看出, 缺少任何一个得分的变种方法的结果都不如使用 3 个得分的结果好. 这证明本文提出的 3 个得分都对根因变更识别有贡献.

表 6 本文方法消融实验结果 (%)

数据集	变种方法	*HR*@1	*HR*@3	*HR*@5
微信变更故障数据集	本文方法	**73.33**	**80.00**	**86.67**
	无差异得分	63.33	70.00	76.67
	无时间得分	56.67	73.33	80.00
	无空间得分	60.00	70.00	70.00
模拟变更故障数据集	本文方法	**74.00**	**84.00**	**88.00**
	无差异得分	62.00	76.00	86.00
	无时间得分	58.00	78.00	86.00
	无空间得分	64.00	74.00	86.00

此外, 从表 6 中还可以发现, 如果缺少时间得分计算模块, 根因变更识别方法在 *HR*@1 上的结果会显著下降 (同比下降 29%). 这一结果进一步证明, 除了考虑变更服务的指标, 软件变更的过程信息也是根因变更识别的重要输入信息之一. 还有一个值得注意的是缺少时间得分计算模块虽然在 *HR*@1 上命中率更低, 但是在 *HR*@3 和 *HR*@5 上的结果却更好. 这是因为故障数据集中存在与告警时间关联性很强, 但是却不是缺陷变更的变更, 这种情况下, 变更时间会成为定位根因变更的干扰. 但融合 3 种得分后, 这种时间得分的干扰会被其他得分抵消, 因而还是能够获得较好的结果.

5.5 问题 8: 参数评估

本节讨论本文方法是否容易受第 4.6.1 节中判断差异是否显著的阈值 λ 的影响. 图 7 展示了不同的 λ 值在微信真实数据集上对本方法的 *HR*@1 和 *HR*@5 的结果影响. 在 DiD 方法中, λ 通常取 0.01–0.2 之间的值. 从图 7 中可以看出, 如果 λ 值设置的过大, 会导致 *HR*@1 和 *HR*@5 的结果下降, 这是因为 λ 过大会导致判定条件宽松, 从而 DiD 更容易产生误报. 根据图中的结果, 本文提出的方法在 $\lambda = 0.15$ 时达到最佳的 *HR*@1 和 *HR*@5. 值得注意的是 $\lambda = 0.15$ 是本文根据微信系统的特性设置的. 在实际应用中, 最佳的 λ 配置需要根据不同系统或数据集的特性来进行设置.

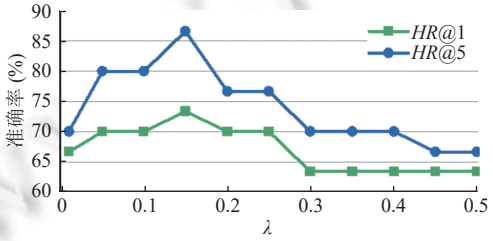

图 7 差异得分计算中的阈值 λ 对本方法准确率的影响

5.6 方法分析开销

本文针对微信生产环境中的典型规模故障 (涉及 50 个服务、200 个变更), 对本文提出的方法进行了工程性能评估, 重点关注内存占用和故障分析时延两项关键指标. 实验结果如表 7 所示. (1) 内存占用: 本文方法在处理时的峰值内存占用约为 2.3 GB, 其中服务依赖图构建模块占用约 0.8 GB, 变更依赖关系构建模块占用约 0.6 GB, 根因变更推断模块占用约 0.9 GB. 在大规模故障场景下, 系统通过增量计算和数据压缩优化, 将内存峰值控制在 4.5 GB 以内, 满足生产环境资源限制要求. (2) 故障分析时延: 在标准配置服务器 (16 核 AMD EPYC 7K62 CPU (2.6 GHz), 32 GB 内存) 上, 本文方法在处理典型规模故障时完成根因变更识别的时延为 28.6 s, 其中服务依赖图构建耗时约 6.8 s, 变更依赖关系构建耗时约 8.4 s, 根因变更推断耗时约 13.4 s.

表 7 方法分析开销

指标类别	测试模块	性能数据
内存占用 (GB)	标准规模故障总开销	2.3
	服务依赖图构建	0.8
	变更依赖关系构建	0.6
	根因变更推断	0.9
故障分析时延 (s)	标准规模故障总开销	28.6
	服务依赖图构建	6.8
	变更依赖关系构建	8.4
	根因变更推断	13.4

6 讨 论

6.1 工业部署案例分析

在本节中, 本文将展示一些微信中的真实根因变更识别案例. 为了保护公司的隐私, 本文对其中一些案例中的部分细节 (例如服务名和具体功能) 进行了匿名处理.

真实案例 I: 开发工程师在服务 S_1 的新版本中引入一段有缺陷的代码, 导致服务在执行该代码段时崩溃, 从而导致服务 S_1 的响应成功率下降. 这一故障是在灰度变更期间由微信的告警系统检测到的. 通过比较服务 S_1 变更前后服务实例的 KPI 曲线, 本文提出的根因变更识别方法能够精准判定该告警是由服务 S_1 的缺陷变更引起的, 并将 S_1 的缺陷变更输出为根因变更得分表的第 1 位. 与运维工程师手动识别根因变更相比, 使用本文的方法根因变更的识别速度提前 61 min 发现了根因变更. 在这个案例里, 当前的缺陷变更检测方法能够识别出服务 S_1 的缺陷变更. 但是, 缺陷变更检测方法同样也将服务 S_1 的上游服务 S_2 的正常变更识别为缺陷变更, 因此 S_2 因异常传播也出现了 KPI 的异常, 没有考虑服务依赖的缺陷变更检测方法会导致误报.

真实案例 II: 开发工程师修改了服务 S_3 某一个函数的返回值类型. 在 S_3 软件变更之后, 服务 S_4 调用服务 S_5 来获取缺陷函数的返回值, 并将该返回值用于调用服务 S_5. 但由于新的返回值的类型与服务 S_5 需要的输入类型不兼容, 导致服务 S_4 调用服务 S_5 返回失败, 从而导致服务 S_4 和 S_5 的响应成功率下降. 在这种情况下, 运维团队首先会检查服务 S_4 和 S_5, 而没有检查到服务 S_3 这种没有出现 KPI 异常的情况. 而本文的方法通过评估故障与变更时间之间的相关性, 能够将服务 S_3 的根因变更排名为最可疑变更 (比运维工程师手动定位根因变更提前 40 min). 由于 S_3 没有出现 KPI 异常, 当前的缺陷变更检测方法 (如 SCWarn) 都未能识别到 S_3 的缺陷变更.

真实案例 III: 开发工程师在服务 S_6 变更时引入了一个内存泄露的代码缺陷. 内存泄露的故障并没有在变更后立即表现出来, 而是在变更完成后 6 h 才出现内存溢出 (OOM), 进而导致影响到用户的访问. 已有的缺陷变更检测方法通常在变更完成后就判定此次变更为正常变更, 因此他们不能处理这种变更完成后才表现异常的缺陷变更. 本文提出的方法在这个案例中没有将 S_6 的变更排名为最可疑的变更, 这是因为系统中存在一个 S_7 的正常变更的变更时间与故障发生时间上关联很高, 所以认为运维工程师应优先检查 S_7. 但考虑了已完成变更的本方法还是能

够将 S_6 的变更排在第 2 位, 这也能够给运维工程师提供参考, 加快运维工程师定位根因变更的速度.

6.2 方法局限性

● 代码级诊断不足: 当前方法主要定位到变更级别 (如服务版本变更), 但未能进一步精确定位到具体代码修改, 但本文认为这种设计是合理的. 在真实工业系统中故障处理流程中存在明确的责任分工: 运维团队负责快速定位并回滚问题变更以恢复服务, 而开发团队随后负责进行深入的代码级诊断以彻底修复问题. 从运维时效性角度看, 生产系统故障响应分为两个独立阶段: (1) 紧急恢复阶段: 需要分钟级定位可回滚的变更版本 (本文焦点); (2) 彻底修复阶段: 开发团队进行小时/天级的代码调试. 这种分工类似于医疗急救中“先止血后手术”的逻辑, 确保服务快速恢复是首要任务. 此外, 从组织安全策略角度看, 运维团队通常无权访问完整代码库, 无法执行代码级诊断. 因此, 本文认为面向变更级的根因定位是填补服务级和代码级诊断之间鸿沟的关键性创新, 它在粒度精确性和运维时效性之间取得了平衡.

● 时间窗口敏感性: 方法依赖 H h 的时间窗口假设, 对于潜伏期较长的缺陷 (如内存泄漏) 可能失效. 真实故障统计显示约 5% 的变更相关故障会在变更 48 h 后才显现, 这类故障难以被当前方法捕获. 特别是在资源耗尽类故障中, 如内存泄漏通常在数天后才表现出明显异常, 而配置不兼容问题可能在系统负载达到特定阈值时才触发. 本文在生产环境中观察到, 将时间窗口扩展至 7 天可以覆盖约 97% 的变更相关故障, 但同时会引入更多无关变更干扰, 导致准确率下降约 5%. 这种时间窗口与准确率之间的权衡需要在不同场景中进行具体调整.

● 跨服务变更传播建模不足: 虽然方法考虑了服务间的直接依赖关系, 但对复杂变更传播路径 (如循环依赖、多跳传播) 的建模仍显不足. 当故障通过非典型路径传播时, 方法的准确性可能下降. 当存在复杂的共享资源竞争 (如多服务共享数据库或缓存) 时, 变更影响可能通过非显式依赖路径传播, 当前方法难以捕捉这种隐式传播机制. 未来工作将探索结合因果推断和动态追踪技术, 构建更完善的故障传播模型.

7 总结与展望

本文的研究意义在于通过对微信系统中真实变更故障数据的实证分析, 揭示了缺陷变更对在线系统生产环境的影响和相关的行为特征, 这为未来的有关研究提供了深入了解变更故障的基础. 同时, 基于实证研究的发现和启发, 我们提出了一种能够高效识别变更故障的根因变更的方法, 为运维工程师提供了更快速、更准确的故障根因变更识别手段. 本文实现了根因变更识别算法的原型, 并使用真实生产环境的变更故障数据集和模拟变更数据集进行了测试. 实验结果表明, 所提方法能够分别以 80% 和 84% 的 $HR@3$ 准确率识别出根因变更, 并且根因变更识别效果显著优于当前最先进的缺陷变更识别方法.

为了进一步推进该领域的研究, 我们将致力于优化和改进根因变更识别方法, 使其能够更好地应用于工业界真实场景. 例如, 目前的 DiD 算法只考虑了变更前后的差异性, 而没有考虑工作负载的周期性. 这可能导致算法误报, 因为工作负载的周期性可能导致 KPI 的变化. 因此, 我们将考虑把历史 KPI 变化纳入算法中. 另外, 在时间得分计算模块方面, 本文将变更后不同时间段的权重视为一致. 然而, 实际上, 离变更时间越近的时间段, 其时间异常得分权重应该更高. 因此, 我们将进一步优化时间得分的权重, 以获得更精确的结果. 此外, 我们还计划继续收集变更故障数据集, 扩大实验规模和数据集范围, 以验证所提方法的普适性和鲁棒性. 通过这些努力, 我们可以进一步提升在线服务系统的可靠性和用户体验, 以应对不断变化的用户需求和市场挑战.

References

[1] Savor T, Douglas M, Gentili M, Williams L, Beck K, Stumm M. Continuous deployment at Facebook and OANDA. In: Proc. of the 38th Int’l Conf. on Software Engineering Companion. Austin: ACM, 2016. 21–30. [doi: 10.1145/2889160.2889223]

[2] The Verge. Facebook says ‘configuration change’ caused some users to be logged out unexpectedly. 2021. https://www.theverge.com/2021/1/23/22245842/facebook-logged-out-configuration-change-ios-app-security

[3] Beyer B, Jones C, Petoff J, Murphy NR. Site Reliability Engineering: How Google Runs Production Systems. California: O’Reilly Media, Inc., 2016. 3–12. [doi: 10.1145/3442381.3449905]

[4] Yu GB, Chen PF, Chen HY, Guan ZJ, Huang ZC, Jing LX, Weng TJ, Sun XM, Li XY. MicroRank: End-to-end latency issue localization

with extended spectrum analysis in microservice environments. In: Proc. of the 2021 Web Conf. Ljubljana: ACM, 2021. 3087–3098.

[5] Yu GB, Chen PF, Li YF, Chen HY, Li XY, Zheng ZB. Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (FSE). San Francisco: ACM, 2023. 553–565. [doi: 10.1145/3611643.3616249]

[6] Yu GB, Huang ZC, Chen PF. TraceRank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems. Journal of Software: Evolution and Process, 2023, 35(10): e2413. [doi: 10.1002/smr.2413]

[7] Li YF, Yu GB, Chen PF, Zhang CF, Zheng ZB. MicroSketch: Lightweight and adaptive sketch based performance issue detection and localization in microservice systems. In: Proc. of the 20th Int'l Conf. on Service-oriented Computing. Seville: Springer, 2022. 219–236. [doi: 10.1007/978-3-031-20984-0_15]

[8] Batta R, Shwartz L, Nidd M, Azad AP, Kumar H. A system for proactive risk assessment of application changes in cloud operations. In: Proc. of the 14th IEEE Int'l Conf. on Cloud Computing (CLOUD). Chicago: IEEE, 2021. 112–123. [doi: 10.1109/CLOUD53861.2021.00025]

[9] Zhang SL, Liu Y, Pei D, Chen Y, Qu XP, Tao SM, Zang Z. Rapid and robust impact assessment of software changes in large Internet-based services. In: Proc. of the 11th Conf. on Emerging Networking Experiments and Technologies. Heidelberg: ACM, 2015. 2. [doi: 10.1145/2716281.2836087]

[10] Xu TY, Jin XX, Huang P, Zhou YY, Lu S, Jin L, Pasupathy S. Early detection of configuration errors to reduce failure damage. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Savannah: USENIX Association, 2016. 619–634.

[11] Li Z, Cheng Q, Hsieh K, Dang YN, Huang P, Singh P, Yang XS, Lin QW, Wu YJ, Levy S, Chintalapati M. Gandalf: An intelligent, end-to-end analytics service for safe deployment in cloud-scale infrastructure. In: Proc. of the 17th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2020). Santa Clara: USENIX Association, 2020. 389–402.

[12] Zhao NW, Chen JJ, Yu ZY, Wang HL, Li JS, Qiu B, Xu HY, Zhang WC, Sui KX, Pei D. Identifying bad software changes via multimodal anomaly detection for online service systems. In: Proc. of the 29th Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 527–539. [doi: 10.1145/3468264.3468543]

[13] Zhang YL, Yang JW, Jin ZQ, Sethi U, Rodrigues K, Lu S, Yuan D. Understanding and detecting software upgrade failures in distributed systems. In: Proc. of the 28th ACM SIGOPS Symp. on Operating Systems Principles. ACM, 2021. 116–131. [doi: 10.1145/3477132.3483577]

[14] Dumitraş T, Narasimhan P. Why do upgrades fail and what can we do about it? Toward dependable, online upgrades in enterprise system. In: Proc. of the 10th Int'l Conf. on Middleware. Urbana: Springer, 2009. 349–372. [doi: 10.1007/978-3-642-10445-9_18]

[15] Li XY, Yu GB, Chen PF, Chen HY, Chen ZK. Going through the life cycle of faults in clouds: Guidelines on fault handling. In: Proc. of the 33rd IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Charlotte: IEEE, 2022. 121–132. [doi: 10.1109/ISSRE55969.2022.00022]

[16] Zhang C, Chen BH, Peng X, Zhao WY. BuildSheriff: Change-aware test failure triage for continuous integration builds. In: Proc. of the 44th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Pittsburgh: IEEE, 2022. 312–324. [doi: 10.1145/3510003.3510132]

[17] Küçük Y, Henderson TAD, Podgurski A. Improving fault localization by integrating value and predicate based causal inference techniques. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Madrid: IEEE, 2021. 649–660. [doi: 10.1109/ICSE43902.2021.00066]

[18] Cai L, Fan YR, Yan M, Xia X. Just-in-time software defect prediction: Literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1288–1307 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5713.htm [doi: 10.13328/j.cnki.jos.005713]

[19] Mehta S, Bhagwan R, Kumar R, Bansal C, Maddila C, Ashok B, Asthana S, Bird C, Kumar A. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In: Proc. of the 17th USENIX Symp. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2020. 435–448. [doi: 10.5555/3388242.3388274]

[20] Zhang L, Qian GQ, Li L. Software stability analysis based on change impact simulation. Chinese Journal of Computers, 2010, 33(3): 440–451 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2010.00440]

[21] Wang XR, Yin KL, Ouyang QY, Wen XD, Zhang SL, Zhang WC, Cao L, Han JX, Jin X, Pei D. Identifying erroneous software changes through self-supervised contrastive learning on time series data. In: Proc. of the 33rd IEEE Int'l Symp. on Software Reliability Engineering (ISSRE). Charlotte: IEEE, 2022. 366–377. [doi: 10.1109/ISSRE55969.2022.00043]

[22] Mahimkar A, de Andrade CE, Sinha R, Rana G. A composition framework for change management. In: Proc. of the 2021 ACM SIGCOMM Conf. ACM, 2021. 788–806. [doi: 10.1145/3452296.3472901]

[23] Stuart EA, Huskamp HA, Duckworth K, Simmons J, Song ZR, Chernew ME, Barry CL. Using propensity scores in difference-in-differences models to estimate the effects of a policy change. Health Services and Outcomes Research Methodology, 2014, 14(4):

166–182. [doi: 10.1007/s10742-014-0123-z]
[24] Chen PF, Qi Y, Zheng PF, Hou D. CauseInfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: Proc. of the 2014 IEEE Conf. on Computer Communications. Toronto: IEEE, 2014. 1887–1895. [doi: 10.1109/INFOCOM.2014.6848128]
[25] Kim M, Sumbaly R, Shah S. Root cause detection in a service-oriented architecture. In: Proc. of the 2013 ACM SIGMETRICS/Int'l Conf. on Measurement and Modeling of Computer Systems. Pittsburgh: ACM, 2013. 93–104. [doi: 10.1145/2465529.2465753]
[26] Ma M, Xu JM, Wang Y, Chen PF, Zhang ZH, Wang P. AutoMAP: Diagnose your microservice-based Web applications automatically. In: Proc. of the 2020 Web Conf. Taipei: ACM, 2020. 246–258. [doi: 10.1145/3366423.3380111]
[27] Lin JJ, Chen PF, Zheng ZB. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Proc. of the 16th Int'l Conf. on Service-oriented Computing. Hangzhou: Springer, 2018. 3–20. [doi: 10.1007/978-3-030-03596-9_1]
[28] Bhagwan R, Kumar R, Ramjee R, Varghese G, Mohapatra S, Manoharan H, Shah P. Adtributor: Revenue debugging in advertising systems. In: Proc. of the 11th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2014). Seattle: USENIX Association, 2014. 43–55.
[29] Lin QW, Lou JG, Zhang HY, Zhang DM. iDice: Problem identification for emerging issues. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Austin: IEEE, 2016. 214–224. [doi: 10.1145/2884781.2884795]
[30] Wang H, Rong GP, Xu YC, You Y. ImpAPTr: A tool for identifying the clues to online service anomalies. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Melbourne: IEEE, 2020. 1307–1311.
[31] Li LQ, Zhang X, He SL, Kang Y, Zhang HY, Ma MH, Dang YN, Xu ZW, Rajmohan S, Lin QW, Zhang DM. CONAN: Diagnosing batch failures for cloud systems. In: Proc. of the 45th IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Melbourne: IEEE, 2023. 138–149. [doi: 10.1109/ICSE-SEIP58684.2023.00018]
[32] Guo XF, Peng X, Wang HZ, Li WX, Jiang H, Ding D, Xie T, Su LF. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. ACM, 2020. 1387–1397. [doi: 10.1145/3368089.3417066]
[33] Kaldor J, Mace J, Bejda M, Gao E, Kuropatwa W, O'Neill J, Ong KW, Schaller B, Shan PJ, 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. Shanghai: ACM, 2017. 34–50. [doi: 10.1145/3132747.3132749]
[34] Viera AJ, Garrett JM. Understanding interobserver agreement: The Kappa statistic. Family Medicine, 2005, 37(5): 360–363.
[35] He ZL, Chen PF, Luo Y, Yan QY, Chen HY, Yu GB, Li FY. Graph based incident extraction and diagnosis in large-scale online systems. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2023. 48. [doi: 10.1145/3551349.3556904]
[36] Kohavi R, Deng A, Frasca B, Longbotham R, Walker T, Xu Y. Trustworthy online controlled experiments: Five puzzling outcomes explained. In: Proc. of the 18th SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Beijing: ACM, 2012. 786–794. [doi: 10.1145/2339530.2339653]
[37] Zhou X, Peng X, Xie T, Sun J, Ji C, Li WH, Ding D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering, 2021, 47(2): 243–260. [doi: 10.1109/TSE.2018.2887384]
[38] Cotroneo D, De Simone L, Liguori P, Natella R, Bidokhti N. How bad can a bug get? An empirical analysis of software failures in the Openstack cloud computing platform. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 200–211. [doi: 10.1145/3338906.3338916]

附中文参考文献

[18] 蔡亮, 范元瑞, 鄢萌, 夏鑫. 即时软件缺陷预测研究进展. 软件学报, 2019, 30(5): 1288–1307. http://www.jos.org.cn/1000-9825/5713.htm [doi: 10.13328/j.cnki.jos.005713]
[20] 张莉, 钱冠群, 李琳. 基于变更传播仿真的软件稳定性分析. 计算机学报, 2010, 33(3): 440–451. [doi: 10.3724/SP.J.1016.2010.00440]

作者简介

余广坝, 博士, CCF 专业会员, 主要研究领域为云计算, 分布式系统可靠性.
陈鹏飞, 博士, 教授, 博士生导师, 主要研究领域为分布式系统, 云计算, AIOps.
唐锡涛, 硕士生, 主要研究领域为分布式系统, 云计算.
郑子彬, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为区块链, 服务计算, 软件可靠性.