自动化、安全性与性能优化在大规模分布式系统中的整合

张志毅[1)]

[1)](华中科技大学 计算机科学与技术学院, 武汉市 430074)

摘 要 随着科技的迅速发展，大规模分布式系统已经成为支持现代应用和服务的核心基础架构。本综述聚焦于如何在这样的复杂环境中实现自动化、提高安全性，并优化性能，以满足日益增长的用户需求。我们综合了相关论文的关键观点，涉及领域包括大规模数据分发、通用分片管理、微服务架构、软件部署工具设计以及异构配置下的系统安全性。首先，我们研究了分布式系统中大规模数据分发的挑战和解决方案，特别关注 Owl 系统如何在 Meta 的私有云环境中实现高效、可扩展的数据分发。其次，我们深入探讨了通用分片管理框架，以解决在异构硬件配置中可能出现的问题，尤其是在地理分布和计划事件方面的挑战。微服务架构作为分布式系统的核心组成部分，也是我们综述的关键焦点。通过分析 Meta 的微服务架构，我们研究了其拓扑结构和请求工作流程的动态性，以及如何通过工具和技术来优化微服务系统的配置和性能。在软件部署方面，我们介绍了 Meta 的 Conveyor 工具，探讨了通用软件部署工具的设计原则，强调了安全在线更新和对大规模复杂模型的支持。最后，我们关注了云系统中异构配置下的系统安全性，通过分析云系统中的异构不安全配置参数，提出了相关的建议和解决方案。本综述旨在为研究人员、工程师和学者提供一个全面的视角，激发对大规模分布式系统中自动化、安全性和性能优化整合的深入研究和创新的兴趣。

关键词 大规模分布式系统；数据分发；通用分片管理；微服务架构；软件部署工具；异构配置；系统安全性

Integration of automation, security and performance optimization in large-scale distributed systems

ZHANG Zhi-Yi[1)]

[1)](School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074)

Abstract With the rapid development of technology, large-scale distributed systems have become the core infrastructure supporting modern applications and services. This review focuses on how to automate, improve security, and optimize performance in such complex environments to meet growing user demands. We synthesize key perspectives from related papers, covering areas including large-scale data distribution, general sharding management, microservice architecture, software deployment tool design, and system security under heterogeneous configurations. First, we examine the challenges and solutions for large-scale data distribution in distributed systems, with a particular focus on how the Owl system enables efficient and scalable data distribution in Meta's private cloud environment. Second, we delve into a common shard management framework to address issues that may arise in heterogeneous hardware configurations, especially challenges with geographic distribution and scheduled events. As a core component of distributed systems, microservice architecture is also a key focus of our review. By analyzing Meta's microservices architecture, we studied the dynamics of its topology and request workflow, and how to optimize the configuration and performance of microservices systems through tools and techniques. In terms of software deployment, we introduced Meta's Conveyor tool, discussed the design principles of general software deployment tools, and emphasized secure online updates and support for large-scale complex models. Finally, we focused on system security under

heterogeneous configurations in cloud systems, and put forward relevant suggestions and solutions by analyzing heterogeneous insecure configuration parameters in cloud systems. This review aims to provide researchers, engineers, and academics with a comprehensive perspective and stimulate interest in in-depth research and innovation on the integration of automation, security, and performance optimization in large-scale distributed systems.

Key words large-scale distributed systems; data distribution; Universal shard management; Microservice architecture; Software deployment tools; Heterogeneous configuration; System security

1 引言

在当今数字化时代，大规模分布式系统已成为支撑各行各业关键应用的核心基础设施。这些系统面临着许多挑战，包括复杂的架构、高度动态的环境和海量的数据处理需求。为了满足这些挑战并提供高效可靠的服务，自动化、安全性和性能优化成为了不可或缺的关键要素。

自动化在大规模分布式系统中扮演着至关重要的角色。随着系统规模的不断增长，手动管理和配置变得越来越困难且容易出错。自动化技术可以帮助我们实现系统的自我配置、自我修复和自我优化，从而提高系统的可靠性和可扩展性。然而，自动化也带来了安全性的挑战。在自动化过程中，必须确保系统的安全性，防止潜在的攻击和数据泄露。

同时，性能优化是大规模分布式系统设计和运维中的另一个重要方面。这些系统需要能够处理海量的请求并在短时间内提供高质量的服务。性能优化涉及到各个层面，包括网络通信、存储系统、负载均衡和数据处理等。通过优化系统的性能，我们可以提高用户体验、降低资源消耗并提升系统的可扩展性。

本文旨在综述自动化、安全性和性能优化在大规模分布式系统中的整合。我们将探讨大规模数据分发的优化、分布式系统中的通用分片管理、微服务架构的可视化与优化、通用软件部署工具的设计与实践和异构配置下的系统安全性等方面的关键问题和挑战，以及当前的研究进展和未来的发展方向。

2 大规模数据分发的优化

首先，文章[1]关注了大规模数据分发，通过解析 Meta 公司的 Owl 系统，研究了在私有云环境中如何实现高效、可扩展的数据分发。这为我们提供了改善数据分发效率和可扩展性的关键见解。

2.1 背景与挑战

在 Meta 的私有云中，向终端主机高效分发大型热门内容是一个日益重要的要求。 三个维度表达了任务的范围：(1) 规模：相同的内容可以被任何地方读取，从少数客户端到全球数据中心中运行的数百万个进程，(2) 大小：要分发的对象范围从 1 MB 到几 TB，以及 (3) 热度：所有客户端可能会在几秒钟内彼此读取一个对象，或者它们的读取可能会分散在几个小时内。 在 Meta，可执行文件、代码工件、AI 模型和搜索索引是此范围内常见分布的内容类型。

分配要求非常严格。首先，内容分发必须快速：人工智能模型的预测价值会随着时间的推移而降低，而缓慢的可执行文件交付会增加停机时间并延迟部署修复程序。我们期望以受读取主机的可用网络带宽或其存储介质的可用写入带宽限制的速率提供数据。

其次，内容分发必须高效。效率的一个维度是可扩展性，即给定数量的服务器可以满足其分发需求的客户端数量。另一个维度是网络使用情况，我们根据传输的字节数和通信局部性来衡量（例如，机架内数据传输比跨区域传输成本更低）。效率的最后一个维度是客户端计算机上的资源使用情况；例如，CPU 周期、内存和磁盘 I/O。我们不仅应该使用尽可能少的资源，还应该根据不同客户的相对重要性进行调整；例如，某些服务受内存限制，而另一些服务则受 CPU 限制或无法写入磁盘。

最后，内容分发必须可靠。可靠性以分发系统在延迟 SLA 内满足的下载请求的百分比来衡量。易于管理的操作是高可靠性的一个经常被忽视的先决条件。在生产环境中，工作负载发生变化，依赖的服务和基础设施可能会出现部分中断，并且依赖项不满足其自身 SLA 的性能故障并不罕见。为了维持较高的分发 SLA，工程师需要快速收到有关此

类事件的警报，并且他们需要清楚了解每种客户端类型的运营状况。最后，他们需要简单的旋钮来在可靠性、速度或效率开始下降时调整行为，以便快速恢复运行状况。

大规模数据分发的挑战包括难以确定委托实现的收益、需要高度灵活且可配置的分发策略来满足广泛变化的分发用例、扩展分发系统以处理超过800PB 的挑战，每天向数百万个客户端进程发送数据，资源效率低下，难以推理整个系统的正确性或效率，以及先前分发系统面临的操作挑战，以及先前分发系统缺乏灵活性，这使得必须使用针对特定用例定制的许多不同的分发解决方案。

2.2 Owl的设计与实现

Owl 有两个基本组件：对等点、链接到使用Owl 下载数据的每个二进制文件的库，以及跟踪器、管理一组对等点的控制平面的专用 Owl 服务。由于容器堆叠和 Twine 容器基础设施对 Owl 的使用，物理主机通常有多个 Owl 对等点。每个跟踪器管理许多对等点：目前有超过 1000 万个 Owl 对等点由 112 个跟踪器管理。此外 Owl 有大约 800 个超级对等点，运行 Owl 库的专用服务提供额外的缓存或执行专门的任务。

2.2.1　临时分发树

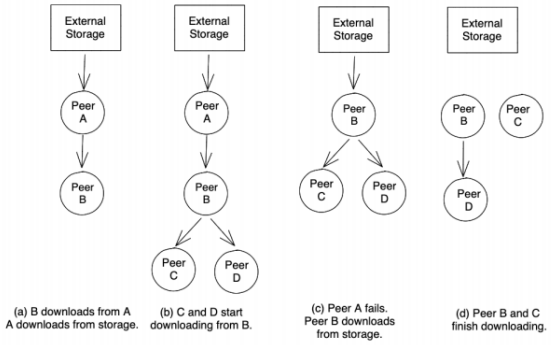


图 1　临时分发树。该图显示了跟踪器如何使用每个块的临时分发树来跟踪哪些对等点缓存了块以及哪些正在下载该块。

临时分发树是跟踪器用来管理每个块下载状态的核心抽象。树的根是外部数据源或缓存块的对等点。有向边指示哪些对等点正在主动从其他对等点下载块；例如，在图 1(a)中，对等点 A 正在从外部存储下载块，而对等点B正在从对等点A下载块。虽然先前的分发系统通常使用树来有效地分发数据，但 Owl 的树特别短暂，因为每个数据块都有自己的树森林，并且节点仅在下载特定块或向另一个对等点提供块时才保留在树中。

在数据平面中，块从根流向叶子；即，沿每个边缘的字节按顺序发送，然后是用于验证完整性的每块校验和。每个对等方在收到新字节后立即将数据转发给其子级。对于大块，这种设计意味着树深度不会强烈影响延迟。叶节点仅看到附加通信跃点的第一个字节延迟，这在数据中心或区域内通常非常小。在图 1(b)中，当节点 C 和 D 请求块时；跟踪器告诉他们从仍在接收数据的 B 获取数据。对等点 B 首先发送其缓存的字节，然后在接收到附加块字节时转发这些字节。

当对等点报告获取数据失败时，跟踪器会删除连接对等点与其父节点的边。如果跟踪器选择一个新的源，它会创建从对等点到该源的边缘。因此，以报告故障的对等点为根的整个子树被移动到树中具有新的父树。当选择新的对等点时，跟踪器避免创建下载周期；即，它不会将同级的后代指定为该同级的新源。树修复对下游节点的影响最小，因为 Owl 在从上次尝试获取最后一个字节后恢复新的下载。在图 1(c)中，对等点 A 发生故障，跟踪器告诉对等点 B 从外部存储中获取剩余字节。对等点 C 和 D 没有注意到这一变化，因为他们继续从 B 下载。

当对等方报告下载成功时，将删除将其连接到其父级的边缘。如果跟踪器要求对等点保留该块，则跟踪器会将对等点添加到已完全缓存该块的节点列表中。由于块可以缓存在多个对等点处，因此块的下载状态是以多个这样的对等点和/或外部数据源为根的临时分发树的森林。在图 1(d)中，对等点 B 和 C 已经下载并缓存了块，而对等点 D 仍在从 B 下载字节。因此，我们在森林中有两棵临时分发树；请求该块的新对等点可能会被定向到这些对等点中的任何一个或外部存储，具体取决于存储桶的选择策略。

2.2.2 跟踪器分片

块的临时分发树被划分为多个跟踪器，其中一个跟踪器的树中的节点充当另一个跟踪器中的子树的根。为了防止该分区树中的循环，跟踪器将不会在以委托对等点为根的树中提供任何对等点以响应委托请求。

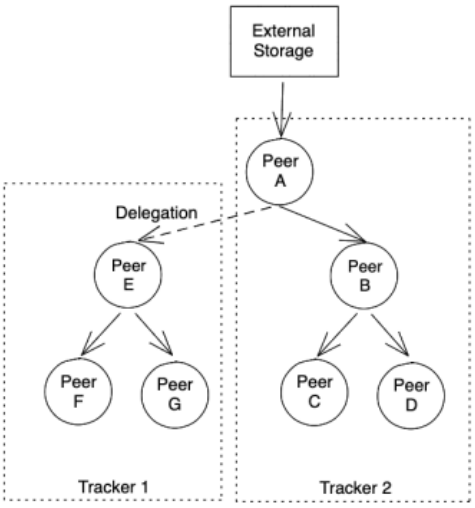


图 2 具有 2 个分片跟踪器的委托。对等点 E 从另一个跟踪器管理的对等点获取块，以减少外部存储上的负载。

图 3 显示了在 2 个跟踪器之间分片的临时分发树。Tracker 2 最初接收来自对等点 A 的 getSource 请求，并指示对等点 A 从外部存储读取块。此时，跟踪器 2 开始向其他分片跟踪器通告它拥有该块。接下来，跟踪器 1 收到来自对等点 E 的 getSource 请求。它在任何对等点上都没有该块，但它知道跟踪器 2 已通告该块。跟踪器 1 向跟踪器 2 发送委托请求，跟踪器 2 选择并返回对等点 A。跟踪器 1 告诉对等点 E 从对等点 A 获取块。当跟踪器 1 收到来自对等点 F 和 G 的后续 getSource 请求时，存储桶的选择策略会优先选择本地-托管对等点，因此这些对等点被引导从对等点 E 获取。如本示例所示，委派提高了分片跟踪器的缓存命中率。如果没有委托，对等点 A 和 E 都将从外部存储中获取数据。通过委派，对等点 A 仅进行一次提取，从而实现了与没有分片时相同的总体缓存命中率。

2.3 优化结果

Owl 将基于临时点对点分发树的分散数据平面与集中控制平面相结合，其中跟踪器服务维护有关对等点、缓存状态和正在进行的下载的详细元数据。在 Owl 中，对等节点是简单的状态机，集中式跟踪器决定每个对等点应从何处获取数据、失败时应如何重试以及应缓存和逐出哪些数据。Owl 跟踪器提供了高度灵活且可配置的策略接口，可以为广泛变化的分发用例定制和优化行为。与之前关于点对点分发的假设相反，Owl 表明集中控制计划并不是可扩展性的障碍：Owl 每天向数百万个客户端进程分发超过 800PB 的数据。与 BitTorrent 和 Meta 使用的先前去中心化静态分发树相比，Owl 将下载速度提高了 2-3 倍，同时支持总共采用 55 种不同分发策略的 106 个用例。

3 分布式系统中的通用分片管理

文章[2]解决了分布式系统采用分片框架时遇到的挑战和障碍。作者认为，现有的分片框架缺乏对大规模互联网服务中普遍存在的地理分布式应用程序的足够支持。他们强调在计划的事件期间维护应用程序可用性的重要性，这是现有分片框架的基本限制。

3.1 分片框架的采用障碍

分片广泛用于扩展应用程序。 尽管花了十年的时间来构建可在不同应用程序中重用的通用分片框架，但其成功程度仍不清楚。 我们试图回答一个基本问题：哪些障碍阻止分片框架被大多数分片应用程序采用？

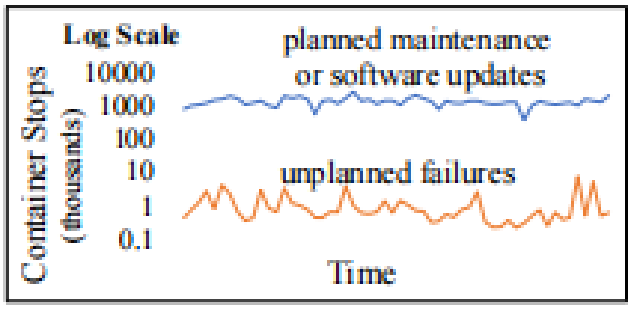


图 3 计划内与计划外的集装箱停靠点

首先，提高可用性是应用程序采用分片框架的主要动力，但现有的分片框架并不能很好地支持这一需求。它们都支持分片故障转移，但它们使用它来处理计划外故障（例如断电或进程崩溃）以及计划事件（例如硬件维护或软件升级）。图 3 显示，在 Facebook，由于计划内事件导致的容器停止频率比计划外故障高约 1000 倍。将计划事件视为故障会将不可用性放大约 1000 倍，因为每次故障转移都会导致一段时间的分片不可用。此外，现有的分片框架无法在不影响正在进行的客户端请求的情况下执行分片迁移（例如，由于负载平衡），这进一步损害了可用性。因此，现有的分片框架通常需要承担部署额外分片副本的成本，以弥补其较低的每个副本可用性。

其次，现有的分片框架无法为大规模互联网服务中普遍存在的地理分布式应用程序提供足够的支持。现有的分片框架无法跨多个区域集群管理器

进行全局协调操作，例如，防止两个地理区域中的两个独立容器重新启动意外地关闭同一分片的两个副本。此外，它们无法跨区域分布分片的副本以获得更好的弹性，无法跨区域迁移分片以获得更好的负载平衡，并且不允许单个分片指定区域放置首选项以获得更好的网络局部性。因此，他们通常要承担将额外分片副本静态部署到多个区域的成本，以弥补对地理分布式应用程序支持的缺乏。

第三，许多应用程序需要高级的 PLB 功能——不幸的是，大多数分片框架都使用手工设计的 PLB 启发式方法，这些方法很容易上手，但随着时间的推移会变得脆弱且难以扩展。因此，很难添加新应用程序所需的新 PLB 功能（例如地理分布式应用程序所需的区域感知分片放置），这进一步阻碍了采用。相比之下，通过向优化问题添加新约束然后使用通用约束求解器求解来引入新的 PLB 功能要容易得多。不幸的是，求解器本身的可扩展性不足以处理近乎实时的 PLB，特别是对于大规模地理分布式应用程序。

第四，现有的分片框架没有考虑到其工作负载的双峰性质，即许多小型应用程序加上一些大型应用程序。小型应用程序更喜欢简单性，而大型应用程序通常采用高度定制的功能。尝试在单个分片框架中实现所有自定义功能会变得笨重且过于复杂。

3.2 解决方案

作者提出了一个用于分片管理器 (SM) 的横向扩展全局控制平面，以解决管理数百万服务器和数十亿分片时的可扩展性挑战。 该控制平面分为多个迷你 SM，每个迷你 SM 管理服务器和分片的子集，旨在通过将大型应用程序划分为不重叠的分区来处理大型应用程序。

此外，该论文还强调了 Facebook 的 Shard Manager 的成功，该管理器目前被运行在超过 100 万台机器上的数百个应用程序使用，约占 Facebook 所有分片应用程序的 54%。这证明了所提出的分片框架的实际适用性和采用。

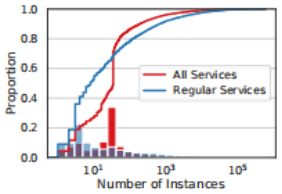
总之，该论文深入探讨了分布式系统中通用分片框架的挑战和解决方案，强调了支持地理分布式应用程序的必要性以及在计划事件期间维护应用程序可用性的重要性。

4 微服务架构的可视化与优化

微服务架构是在许多组织中构建和操作分布式应用程序的新颖范例。与单体应用程序相比，这种范例改变了分布式应用程序的构建、管理和操作方式的许多方面。它引入了需要解决的新挑战，并需要改变对以前众所周知的挑战的假设。但是，如今，大规模微服务架构的特征在组织外部是不可见的，从而抑制了研究机会。最近的研究仅提供了部分概览，并且仅代表了单个设计点。本节通过描述 Meta 的微服务架构，丰富了我们对大规模微服务的理解。它重点关注以前未报告（或报告不足）的方面，这些方面对于开发和研究使用微服务拓扑或请求工作流跟踪的工具非常重要。[3]

4.1 拓扑特性

规模以百万个实例来衡量。截至 2022 年 12 月 21 日，微服务拓扑包含 18500 个活动服务和超过 1200 万个服务实例。除去不合适的服务，共有 7400 个服务和 1120 万个实例。



(a) CDF

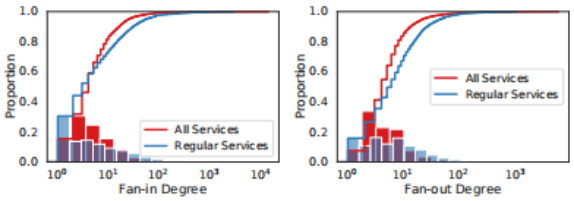
	# Instances	# Instances (Regular)
Min	1	1
Med	34	8
Avg	668	1,504
P99	9,220	31,306
Max	557,156	557,156

(b) Statistics

图 4　服务 ID 复制因素。直方图显示在 CDF 下方。当曲线重叠时，颜色会混合在一起。

实例计数是由一些高度复制的服务造成的。图 4 显示，不合适的服务极大地扭曲了实例计数。值得注意的是，ML 调度程序被复制超过 270000 次，占所有实例的 2.2%。当排除这些服务时，中位服务的复制因子仅为 8，第 99 个百分位为 31306。前端服务 www 是重复次数最多的服务（557000 个实例，占所有实例的 4.6%），因为它处理大多数传入请求。

服务是稀疏互连的。通过使用边缘连接至少一次相互通信的服务来构建服务依赖关系图。（依赖关系图与 OpenTelemetry 或 Jaeger 构建的依赖关系图类似，不同之处在于它是根据服务历史记录数据集的一部分构建的，该数据集捕获服务之间的通信，而不是原始跟踪。）连接服务的边有 393622 条，数量要小得多比完全连接的拓扑（185002 或 3.42 亿个边）要多。



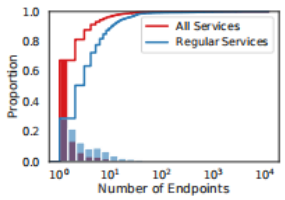
(a) Fan-in CDF (b) Fan-out CDF

Statistic	Fan-in	Fan-out	Fan-in (Reg)	Fan-out (Reg)
Min	1	1	1	1
Median	4	4	3	6
Average	14	12	19	15
P99	86	101	324	158
Max	14,084	5,865	2,968	1,069

(c) Statistics

图 5 服务扇入和扇出

服务调用服务的次数多于调用其他服务的次数。继续查看依赖关系图，图 5 显示了 CDF 以及有关服务扇入（调用它们的服务的数量）和扇出（它们调用的服务的数量）程度的统计信息。扇入和扇出的中位数相同，但平均和最大扇入大于扇出（14 与 12 以及 14084 与 5865）。通过删除与不合适服务连接的所有边来排除不合适的服务，可以减少中值扇入，但增加中值扇出。排除不合适的服务也会增加 99.9 百分位数并降低最大扇入和扇出值。



	# Endpts	# Endpts (Regular)
Min	1	1
Med	1	2
Avg	5	14
P99	26	61
Max	11,359	11,359

(a) CDF (b) Statistics

图 6 服务公开的端点数量

大多数服务都很简单，仅公开几个端点。图 6 显示了 CDF 和服务端点的统计信息。大多数服务不会公开很多端点（中位数：1，P99：26），并且排除不合适的服务不会使统计数据发生太大变化。公开 11359 个端点的服务是 www，它是一种用于许多业务用例的前端服务。它作为单个二进制文件部署在微服务架构出现之前的大型、精心设计的代码库中。

服务复杂性遵循幂律分布。服务复杂性通过服务中唯一端点的数量来衡量，遵循幂律分布（α =2.23，R2=0.99），这表明大多数服务都很简单，并且具有更多的长尾复杂的服务。幂律不适用于其他复杂性度量。尽管存在更复杂服务的长尾，但服务依赖关系图并不遵循幂律分布 (R2=0.62)。这意味着具有更多端点的服务并不比具有更少端点的服务更多地连接到拓扑。虽然存在一些高度复制的服务，但实例计数的总体趋势也不遵循幂律分布（R2 =0.25）。

用于编写服务的十六种不同语言。服务可以用多种编程语言编写。目前 Meta 使用 16 种不同的编程语言，按代码行数衡量，最流行的是 Hack（PHP 的一个版本）。其他流行的语言包括：C++、Python 和 Java，其余语言形成长尾。

4.2 请求工作流特征

使用不同配置文件收集的跟踪来分析各个请求工作流的服务级别属性。首先讨论走线的一般特征，例如尺寸和宽度。然后，分析单个迹线的特定元素是否可以预测代表相同高级行为的其他迹线的属性。与任何大规模跟踪基础设施一样，由于记录丢失、速率限制和非仪表化服务，跟踪对请求工作流的可见性可能会受到限制。

方法：对于所有实验，使用 2022 年 12 月 21 日从三个配置文件收集的跟踪记录，用于监控重要的高级业务功能。使用几个配置文件和一天可以避免一些因素，否则会掩盖结果的可解释性：使用许多采样策略和改变服务行为的代码更新来分析跟踪的影响。关注重要的配置文件会增加跟踪代表其工作流程的可能性：他们访问的服务可能会准确地传播上下文并使用描述性服务 ID 和端点名称。总体而言，分析了 650 万条痕迹，占所有 Canopy 配置文件在 2022 年 12 月 21 日收集的痕迹的 0.5%。尽管没有报告这些结果，但在改进实验时，观察到了与 2022 年 12 月 21 日相邻几天的结果相似的趋势。

对于可预测性实验，对 Ingress ID（定义为服务 ID 和入口端点名称的组合）是否可以跨多个跟踪预测其子项的属性进行事后分析。选择使用 Ingress ID 是因为它们在迹线中很容易获得，通常是理解迹线行为的主要手段，并且与位置无关，因此不需要从拓扑中的不同（未知）深度开始对齐迹线。对于预测实验，不考虑迹线的全局特征，例如大小或宽度，因为由于速率限制或丢失的迹线记录，不能保证它们具有可比性。在可预测性部分中，当提到父母和孩子时，指的是 Ingress ID，例如“unique children”。

在考虑服务名称的实验中省略了推断调用，因为推断服务的名称通常是未知的。此外，省略了在配置文件中发现次数少于 30 次的入口 ID，以便为其余端点计算有意义的统计数据。

主要发现总结如下：

测量有关其包含的服务块数量的跟踪（服务块

表示服务执行的时间间隔；同一服务的重复调用显示为多个服务块。) 跟踪大小根据工作流的高级行为而有所不同，但大多数都很小（仅包含几个服务块）。跟踪通常很宽（服务调用许多其他服务），深度很浅（调用者/被调用者分支的长度）。

根入口 ID 不预测跟踪属性。在父/子关系级别，父项的 Ingress ID 可以预测父项将在至少 50%的执行中调用的子项 Ingress ID 集。但是，它不能很好地预测父级的 RPC 调用总数或 RPC 调用之间的并发性。将子集的 Ingress ID 添加到父 Ingress ID 可以更准确地预测 RPC 调用的并发性。

观察到跟踪中的许多调用路径由于速率限制、丢弃记录或未检测的服务而过早终止。这些调用路径中很少有可以重建的（已知在数据库处终止的调用路径），而大多数是不可恢复的。更深层次的调用路径被不成比例地终止。

5 通用软件部署工具的设计与实践

本节展示 Meta 的软件部署工具 Conveyor，以及通过管理 30000 多个部署管道获得的宝贵数据，这些管道在 Meta 上的数百万台机器上部署各种服务。描述 Conveyor 支持实现普遍覆盖的广泛部署场景。在 Meta，在通过容器部署的所有服务部署管道中，97%采用完全自动化部署，无需人工干预：55%采用持续部署，在通过自动化测试后立即将每个代码更改部署到生产环境，其余 42%按固定时间表（主要是每天或每周）自动部署，无需手动验证。我们重点介绍 Conveyor 的几个显着功能，包括用于降低硬件成本的安全就地更新、分析代码依赖性以防止错误发布，以及大规模部署复杂 ML 模型的能力。[4]

5.1 启用就地更新

软件部署方法会影响硬件成本。就地更新方法会在同一台计算机上重新启动现有任务以运行新的可执行文件。相比之下，镜像方法也称为红黑或蓝绿部署，首先使用新的可执行文件启动一个新作业（通常在其他机器上），逐渐将流量从旧作业重定向到新作业，最后关闭旧作业。尽管这种方法更安全，因为如果更新失败，流量可以快速重定向回旧作业，但它需要额外的硬件来并行运行旧作业和新作业。考虑每三个小时向 50 万台机器部署 FrontFaaS 的示例。简单的镜像方法需要 500K 的额外机器，这是不可接受的。一种优化是将其分成许多小作业，并利用镜像一次更新一个小作业。然而，这种方法会导致快速回滚能力的丧失并使作业自动缩放变得复杂。尽管进一步优化是可能的，但最终的解决方案不一定比就地更新更便宜、更简单或更通用。

尽管有节省硬件的好处，但由于难以确保部署安全，就地更新方法缺乏现有部署工具的广泛支持。下面，介绍如何通过共同设计部署工具（Conveyor）和集群管理器（Twine）来实现安全实用的就地更新。

Conveyor 和 Twine 之间的协同控制。在部署操作的每个阶段，Conveyor 指示 Twine 更新特定数量或百分比的任务，表示为 Nbig，然后等待一段时间收集全面的运行状况信号，然后再进入下一阶段。然而，Twine 不会一次性更新所有 Nbig 任务。相反，它在一批中仅更新 Nsmall 任务，其中 Nsmall 比 Nbig 小得多，以避免同时丢失太多任务。然后，Twine 检查每个任务的活跃度，然后再继续更新下一个 Nsmal 任务。与其他集群管理器类似，Twine 的活性检查是初级的，仅验证单个任务是否正常运行。然而，它无法检测微妙的问题，例如与旧代码相比新代码的内存回归，这是由 Conveyor 的全面运行状况检查处理的。

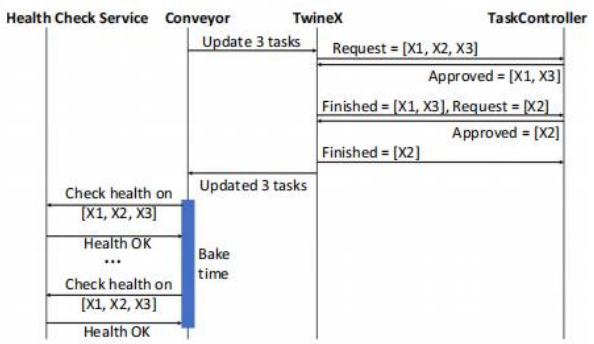


图 7　更新 KVStore 的第 1 阶段

可插入任务控件。大多数服务可以使用每服务常 Nsmall，并且对要更新的特定任务没有限制。然而，分片键值存储提供了一个需要更精确控制 Nsmall 和要更新的特定任务的服务示例。图 7 进一步说明了此类服务如何实现自定义 TaskController 以确保安全的就地更新。一般来说，TaskControl 使服务能够出于各种原因精确控制任务更新，而不仅限于数据分片的可用性。例如，FrontFaaS 利用 TaskControl 来最大限度地提高其部署速度。

硬件故障和计划维护。当多个任务同时进行就地更新时，存在将服务的可用容量降低到不健康水

平的风险。仅仅通过部署管道控制更新速度是不够的，因为某些任务可能由于机器故障或计划维护而处于不健康状态，而 Conveyor 不知道这一点。为了解决这个问题，服务所有者可以向 Twine 告知预算，该预算指示可以因任何原因（例如任务更新、硬件故障或计划维护）而离线的任务的最大数量或百分比。如果预计超出预算，Twine 会暂停任务更新。

零停机热插拔。边缘数据中心的路由服务将面向用户的流量转发到数据中心，并保持与用户设备的实时 HTTPS 连接。天真地重新启动更新任务会导致用户面临的错误。Twine 提供了同主机热插拔功能来解决这个问题。它首先在同一台机器上启动一个新任务，该任务绑定到与旧任务相同的 TCP 端口。然后，新任务和旧任务在 eBPF 的帮助下相互协作，将旧任务的实时连接移交给新任务。热插拔的一个限制是它需要容器配置足够的内存来启动两个任务，这也是它没有被普遍使用的原因。

如果没有集群管理器的支持，所有上述具有就地更新的部署场景都无法正确实施。这是现有部署工具主要使用镜像方法的一个关键原因，因为它们所依赖的集群管理器不提供就地更新所需的功能。

5.2 代码依赖性分析以防止错误发布

monorepo 将组织的许多项目的代码存储在单个存储库中，促进代码重用，但也会导致代码依赖性增加。例如，ServiceRouter 库被编译到 Met 中几乎每个服务中，它可能依赖于高性能数据结构库，而高性能数据结构库又可能依赖于分析库，等等。在 monorepo 设置中，每当提交库的新版本时，依赖于该库的任何服务都将使用新版本自动编译。因此，服务的所有者甚至可能不知道他们的服务受到共享库中的错误的影响。为了解决这个问题 Conveyor 提供了坏包检测器 (BPD)。如果库开发人员发现库中存在错误，他们可以向 Conveyor 报告。然后，BPD 利用构建系统提供的代码依赖图来识别并取消使用有问题的库版本构建的所有服务可执行文件的发布。

准确的代码依赖性分析对 BPD 提出了挑战，因为它需要在误报和误报之间找到适当的平衡。实现完美的覆盖需要考虑服务的所有可能的直接和间接依赖性，这通常是不切实际的。为了达到平衡 BPD 目前跟踪 14 个依赖级别。生产数据显示，大约 14%的待部署可执行文件因 BPD 而失效。 这凸显了处理相关代码中的错误的重要性。

5.3 机器学习模型部署

鉴于机器学习应用程序数量的迅速增加，用于推理的机器学习模型的部署已成为一个重要问题。虽然现有的部署工具通常不处理模型部署，但 Conveyor 已针对这一需求进行了专门增强，目前约 44%的管道用于模型部署。下面，描述 Conveyor 对模型部署的支持。

通过配置更改进行部署。在 Conveyor 的第一个模型部署实现中，模型更新和可执行更新共享同一个管道，需要 Twine 重新启动容器。然而，由于模型更新可能比推理可执行文件的更新更频繁，因此频繁的容器重启会导致频繁损失用于请求服务的昂贵的 GPU 容量。此外，由于模型通常包含千兆字节 (GB) 的数据，因此在重新启动期间加载 GB 数据所需的时间可能很长。

为了解决这个问题，一些推理服务使用两个正交管道。一个管道通过 Twine 部署推理可执行文件，而另一个管道通过 Meta 的配置管理系统 Configerator 部署模型数据。为了跟踪模型更新，为模型提供服务的所有任务都会订阅指定要提供的模型的当前版本的配置。当模型的新版本可用时，Conveyor 不会同时将其暴露给所有任务，而是指示 Configerator 按照部署管道分阶段逐步将新版本暴露给任务。配置器利用数据分布树以可扩展的方式通知任务。一旦任务收到新模型版本的通知，它们就会利用 Owl[1]（一种点对点数据分发系统）来获取新模型。在仍然服务实时请求的同时，任务将新模型逐块合并到旧模型中，而不消耗额外的内存，因为它永远不会同时在内存中保留两个模型的完整副本。总体而言，Conveyor 通过分阶段发布确保模型的安全部署，并通过配置更改进行精心管理。

Conveyor 执行通用配置更改的分阶段部署的能力超出了其在 ML 模型更新中的用途。输送管道被广泛利用，以确保各种配置更改的安全部署。

相互依赖的服务的同步部署。一些机器学习模型太大，无法容纳一台机器的内存，因此它们被划分为相互依赖的分片，每个分片都包含多个副本，以实现容错和吞吐量。每个分片都映射到不同的作业，通常第一个分片充当聚合器来组合其他分片的结果。但是，独立更新不同的分片可能会导致兼容性问题，因为组合不同版本分片的输出会产生不正确的结果。为了确保兼容性，第一个分片的副本被配置为仅接收来自相同版本的其他分片的副本的输出。这种设计要求 Conveyo 对不同分片执行锁步

部署，以避免部署过程中的容量损失。例如，每个分片的 5%的副本会同时更新，5%的客户端流量会定向到新版本，然后再继续更新每个分片的 10%的副本，依此类推。

亲子管道。Meta 的 ML 推理系统使用大约 10 种不同的推理可执行文件为数万个 ML 模型提供服务。每个模型及其推理可执行文件都通过单独的管道进行部署。由于许多模型共享相同的推理可执行文件，因此更新一个可执行文件可能会导致数千个管道同时启动新版本。这不仅会导致 Conveyor 和 Twine 上的负载峰值，还会增加推理可执行文件中未检测到的错误同时影响许多模型的风险。

虽然现有的部署工具单独管理每个管道，但 Conveyor 通过在管道之间设置父子关系来协调共享公共工件的跨管道的发布。具体来说，每个推理可执行文件均由父管道管理，其中包括复杂的测试但不包括部署操作，而可执行文件所服务的模型的管道充当其子管道并包括部署操作。当更新 ML 模型而不修改可执行文件时，仅执行相应的子管道，而不涉及父管道。但是，更新可执行文件时，首先执行父管道。如果成功，所有相应的子管道将以随机延迟执行，以避免同时启动它们并导致系统过载。此外，它可以配置为首先执行不太重要模型的子管道子集，以帮助检测可执行文件的问题。

6 异构配置下的系统安全性

随着异构硬件的日益普及以及在线重新配置的需求不断增加，对异构配置的需求也越来越大。然而，允许不同的节点具有不同的配置可能会在这些节点通信时导致错误，即使每个节点的配置使用有效值。

虽然许多分布式系统最初是在所有节点共享相同配置的假设下设计的，但由于两个原因，异构配置变得越来越流行。首先，异构硬件自然需要异构配置来实现最佳性能。其次，即使对于同构系统，有时我们也需要在运行时更改其配置以适应工作负载，但使用新配置重新启动整个系统可能会造成太大的破坏性。为了解决这个问题，有几种方法可以通过重新启动这些节点或利用应用程序 API 来逐步更改节点子集的配置，直到所有节点都已更改新配置。这两种情况都可能导致不同的节点具有不同的配置，无论是长期还是短期。

然而，异构配置如果使用不当可能会导致系统故障。例如，如果一个节点配置为加密其通信通道，而另一节点不解密消息，那么毫无疑问，通信将失败。此类错误与无效配置值引起的配置错误不同：在我们的例子中，两个配置值（即使用和不使用加密）都是有效的；该问题是由具有不同配置的两个节点相互通信引起的。

然而，在异构硬件配置中如何测试和提高系统的安全性？

6.1 测试系统安全性

在异构硬件配置中测试系统的安全性可能具有挑战性。解决这一挑战的一种方法是利用现有的单元测试来识别异构不安全配置参数。通过在这些测试中向不同节点分配不同的配置，可以测试某些配置参数的效果并识别潜在的漏洞。

为了实现这一目标，Ma 等[5]开发了一个名为 ZebraConf 的框架，它提供了一个 API，可以以最小的努力修改目标应用程序，并提供减少运行测试数量的策略。ZebraConf 已经在 Flink、HBase、HDFS、MapReduce、YARN 等流行的开源应用程序上进行了评估，并成功识别了这些应用程序中的 41 个异构不安全配置参数。

ZebraConf 结合了多种技术来解决运行大量测试的挑战，包括“预运行”单元测试来识别每个单元测试中每种节点类型使用哪些配置参数，以及引入池测试来一起测试多个参数 通过一个单元测试。这些技术有助于提高准确性并减少测试的运行时间。

从系统架构来看，ZebraConf 由三个关键组件组成：TestGenerator、TestRunner 和 ConfAgent。TestGenerator 确定要运行哪些单元测试以及每个单元测试要使用哪些异构配置，而 TestRunner 测试单元测试是否报告给定异构配置和任何相应同构配置的错误。

总之，利用现有的单元测试并利用 ZebraConf 等框架可以成为在异构硬件配置中测试系统安全性的有效方法。

6.2 提高系统安全性

为了提高异构硬件配置中系统的安全性，可以实施多种策略。首先，必须识别和解决异构不安全配置参数，这些参数在具有不同配置的不同节点相互通信时可能导致错误。这可以通过利用像 ZebraConf 这样的框架来实现，该框架提供了一个 API，可以以最小的努力修改目标应用程序，并提

供减少运行测试数量的策略。

此外，重要的是要确保配置值得到不同节点的同意，并且不必要的参数不会暴露给最终用户，因为它们可能充当最终用户依赖应用程序实现细节的侧通道。此外，全系统单元测试可以提供有效的解决方案来识别和解决异构不安全配置参数，因为它们非常类似于真实的分布式设置，并且可以帮助防止测试过程中的误报和误报。

通过实施这些策略，管理员可以提高异构硬件配置中系统的安全性，并减少由于不同节点具有不同配置而导致错误的可能性。

7 总结

本文对自动化、安全性和性能优化在大规模分布式系统中的整合进行了综述。探讨了如何优化大规模数据对象的分发，介绍了背景与挑战、Owl 的设计与实现以及优化后的结果；研究了分布式系统中的通用分片管理，尤其是分片框的采用障碍和解决方案；研究了 Meta 微服务架构的拓扑特性和请求工作流特征；介绍 Meta 的部署工具 Conveyor 的几个显着功能，包括用于降低硬件成本的安全就地更新、分析代码依赖性以防止错误发布，以及大规模部署复杂 ML 模型的能力；并且研究了异构配置下的系统安全性。然而，仍然存在一些挑战和未解决的问题，如自动化与安全性之间的平衡、性能优化的复杂性等。未来的研究应该致力于解决这些问题，并探索新的方法和技术，以进一步推动大规模分布式系统的发展和创新。

致 谢 感谢施展老师、胡燏翀老师和童薇老师关于数据中心技术这门课程的悉心教导。课堂知识的讲述、论文的研讨汇报以及老师同学提出的意见都使我受益良多。

参 考 文 献

[1] Jason Flinn, Dou Xian-Zheng, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Zhou Fang. Owl: Scale and Flexibility in Distribution of Hot Content//Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. Carlsbad, CA, 2022: 1-15.

[2] Sangmin Lee, Guo Zhen-Hua, Omer Sunercan, Ying Jun, Thawan Kooburat, Suryadeep Biswal, Chen Jun, Kun Huang, Yatpang Cheung, Zhou Yi-Ding, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Tang Chun-Qiang. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications// Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. Virtual Event, Germany, 2021: 553–569.

[3] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta’s microservice architecture: Analyses of topology and request workflows//Proceedings of the USENIX Annual Technical Conference. Boston, USA, 2023: 419-432.

[4] Boris Grubic, Wang Yang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Tang Chun-Qiang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta// Proceedings of the USENIX Symposium on Operating Systems Design and Implementation. Boston, USA, 2023: 325-342.

[5] Ma Si-Xiang, Zhou Fang, Michael D. Bond, and Wang Yang. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems//Proceedings of the European Conference on Computer Systems. Virtual, 2021: 410–425.

附录 1

问题 1：持续部署的概念？

软件部署是指将软件应用程序或系统的新版本、更新或变更从开发环境或测试环境部署到生产环境中，以便用户可以访问和使用更新后的软件。这个过程涉及将软件代码、配置文件和其他相关资源部署到目标服务器或设备上，并确保新版本的软件能够正常运行并与现有系统兼容。软件部署通常需要考虑安全性、稳定性和性能等方面，以确保部署过程不会影响现有的业务运行。

持续部署（Continuous Deployment）是软件开发和交付过程中的一种实践方法，旨在实现快速、自动化和可靠地将应用程序的变更部署到生产环境中。在传统的软件开发模式中，开发团队通常会经历漫长的开发周期，然后将代码交付给运维团队进行部署。这种方式存在许多挑战，如部署延迟、人为错误、手动操作等，限制了软件交付的速度和质量。持续部署就是为了解决这些挑战。它建立在持续集成（Continuous Integration）的基础上，强调频繁地将代码变更集成到主干代码库，并通过自动化的流程进行构建、测试和部署。持续部署的目标是实现每次代码提交都可以自动部署到生产环境，从而实现快速的软件交付和反馈循环。

持续部署可以带来许多好处，如更快的交付速度、更高的开发效率、更快的问题解决和用户反馈循环等。然而，它也需要具备良好的测试覆盖率、自动化测试和部署流程、稳定的基础设施和监控等条件，以确保安全和可靠的部署。

问题 2：为什么阿里、腾讯等大厂要及早发布，经常发布？

1、 快速反馈和迭代。通过及早发布和频繁发布，这些公司能够更快地获取用户的反馈和数据，了解用户需求和行为，从而更好地优化产品和服务。他们相信持续的迭代和改进是实现产品成功的关键。
2、 快速响应市场需求。科技行业竞争激烈，市场需求和用户偏好在不断变化。通过频繁发布，这些公司能够更快地推出新功能、新产品或对现有产品进行改进，以满足市场需求，保持竞争优势。
3、 降低风险和快速修复问题。及早发布和频繁发布可以帮助这些公司更早地发现和修复问题。通过持续的测试和监控，他们能够及时发现潜在的缺陷、性能问题或安全漏洞，并迅速采取措施解决，从而降低风险和对用户的影响。
4、 提高团队效率和协作。频繁发布需要团队建立高效的开发、测试和部署流程。这促使团队更加注重自动化、持续集成和持续交付等最佳实践，提高开发效率和团队协作能力。
5、 增强用户体验。通过及早发布和频繁发布，这些公司能够更快地为用户提供新功能和改进，提升用户体验，满足用户的期望，并留住用户。

这些公司在发布前会进行充分的测试和质量保证，确保产品的稳定性和可靠性。他们注重用户反馈和数据驱动的决策，以确保每次发布都是有价值的，并能够快速修复任何问题。这种发布策略需要结合良好的技术基础设施、自动化流程和团队协作，才能实现成功。