MicroFog：用于在联邦雾环境中可扩展部署基于微服务的物联网应用的框架

SamodhaPallewatta，VassilisKostakos 和Rajkumar Buyya

澳大利亚墨尔本大学计算与信息系统学院云计算与分布式系统（*CLOUDS*）实验室

文章信息

关键词：雾计算 微服务
应用放置 物联网 微服务
组合

摘要

微服务架构（MSA）正迅速成为开发大规模物联网应用的热门选择，这些应用将部署在分布式且资源受限的雾计算环境中。作为一种云原生应用架构，微服务的真正优势在于其松耦合、可独立部署且可伸缩的特性，这使其能够在联邦雾云集群中实现分布式放置和动态组合。因此，有必要开发新型放置算法，利用微服务的这些特性来提升应用性能。然而，现有雾计算框架在多个方面存在不足，因此无法支持集成此类放置策略，包括在多雾多云环境中进行MSA 应用放置与部署、跨多个分布式集群动态组合微服务、框架在联邦环境中的可伸缩性，以及对异构微服务应用部署的支持等。为此，我们设计并实现了 MicroFog，这是一个兼容 Docker、Kubernetes 和 Istio 等云原生技术的雾计算框架。MicroFog 提供了一个可扩展且可配置的控制引擎，用于执行放置算法并在联邦雾环境中部署应用。此外，MicroFog 对容器编排和微服务动态组合提供了充分抽象，使用户能够轻松集成新的放置策略并评估其性能。我们通过多个用例验证了 MicroFog 框架的各项能力，例如其设计与部署架构的可伸缩性和灵活性，以及确保在分布式雾云环境中部署和组合微服务的能力。实验还展示了MicroFog 集成和评估新型放置策略与负载均衡技术的能力。为此，我们集成了多种微服务放置策略，以展示 MicroFog 支持在联邦环境中对微服务进行水平扩展式放置、服务发现和负载均衡的能力，从而将应用服务响应时间最多缩短 54%。

arXiv:2302.06971v1 [cs.DC] 14 Feb 2023

1. 引言

物联网（IoT）正在快速发展，数量不断增长、种类日益丰富的互联设备产生了海量数据，这些数据涉及广泛的智慧应用领域，例如智慧城市、智慧医疗、工业物联网和智慧交通等。因此，物联网应用开发正采用微服务架构（MSA），以支持物联网应用的快速演进，进而构建物联网生态系统。作为一种云原生应用架构，微服务架构将应用构建为一组称为微服务的模块，这些模块可独立部署和扩展[1]。微服务采用 Docker 等技术进行容器化，并借助 Kubernetes 等容器编排平台以及 Istio 等服务网格技术进行动态组合，从而确保部署在分布式计算资源上的微服务之间实现无缝连接。

与此同时，雾计算正成为一种强大的分布式计算范式，可用于托管对延迟敏感且带宽需求高的物联网应用。雾计算利用位于连接物联网设备与集中式云数据中心的路径中的计算、网络和存储资源 [2]，将类似云的服务延伸至网络边缘。随着物联网应用的日益普及，将大量数据发送到集中式云会导致高延迟和带宽拥塞。此外，分布式雾资源为物联网应用提供了所需的位置感知、数据安全、移动性感知和可伸缩性，同时地理分布式用户也希望能够泛在访问应用服务 [3]。

尽管使用分布式雾计算资源是解决这一问题的一种方案，但雾资源的资源受限特性是其主要缺点，而通过地理分布式雾集群与云数据中心的联邦化可以克服这一缺点。这包括协同使用分布式雾计算集群/数据中心和云数据中心来进行应用程序放置，以满足其需求并符合服务质量要求 [4]。这种方法着眼于扩展混合云，将多个雾基础设施提供商（FIP）提供的雾计算资源纳入其中，并保持不同环境之间的无缝连接，从而实现尽可能优异的性能 [5]。此外，微服务的云原生特性使其非常适合大规模物联网应用的此类放置，由此催生了渗透计算等新型范式。渗透计算提出了物联网、微服务架构（MSA）与雾计算的融合，使微服务能够在混合雾云环境中动态迁移和组合 [6]。

ppallewatta@student.unimelb.edu.au (S. Pallewatta) ORCID:

为了充分发挥雾计算环境中微服务架构（MSA）的潜力，开发高效的放置算法至关重要。因此，针对基于微服务的物联网应用放置算法的设计、开发与评估研究正受到广泛关注。现有文献包含一些研究，重点关注通过水平扩展进行微服务放置，以满足吞吐量、可靠性和延迟等服务质量参数 [7, 8, 9, 10],位置感知放置 [11], 等；这些研究将相互连接的微服务放置在分布式资源上。然而，在企业级应用这些算法之前，还需要进行广泛且准确的评估与验证[12]。

可通过数值评估 [11, 8], 模拟器 [13, 10, 14] 以及小规模测试平台上的真实部署 [7, 15]来评估放置策略。云计算相关策略可通过租用模式，使用 Amazon AWS、Google Cloud 等商业服务提供商通过基础设施即服务（IaaS）提供的云基础设施进行评估。由于缺乏此类雾计算平台，雾应用放置策略主要通过数值评估和模拟器进行评估。尽管已有若干用于管理雾资源的真实框架 [16, 17], , 但它们在基于微服务的物联网应用放置方面存在局限。这些框架缺乏对联邦式雾与云数据中心之间微服务动态组合的支持，难以集成分布式放置策略，与开源云原生技术的兼容性不足，对异构的基于微服务的应用支持有限，且设置和原型开发支持不便等。为克服这些局限，我们提出 MicroFog：一个易于配置的软件框架，用于在联邦雾云环境中进行基于微服务的应用放置。物联网应用开发者、雾基础设施提供商以及雾计算研究人员可使用 MicroFog 创建、集成并评估新颖的放置策略，以部署和管理基于微服务的物联网应用。MicroFog 使用户能够创建充分发挥微服务架构潜力的放置方法，从而提升应用的服务质量。

MicroFog 提供了一个可配置的控制引擎，可通过分布式或集中式方式执行放置策略，并在由 Kubernetes 和 Istio 管理的雾与云资源集群中部署容器化微服务。MicroFog 抽象了 Kubernetes 和 Istio 资源部署（即 Pod、服务、虚拟服务、网关等），同时支持集成新颖的放置算法和负载均衡策略。此外，MicroFog 通过支持服务发现和负载均衡，确保分布于地理位置分散的多雾多云环境中的微服务能够动态组合。

我们工作的主要贡献如下：

- 提出了一种可扩展框架，用于在联邦雾计算与云计算环境中部署和管理基于微服务的物联网应用程序。该框架由多个组件组成，包括用于执行放置算法和进行应用部署的控制引擎（MicroFog-CE）、用于存储所需元数据的数据存储、监控组件和日志组件。

- MicroFog-CE 被设计并开发为一种易于配置的微服务，支持不同的运行模式（集中式与分布式）、应用放置模式（周期性与事件驱动），并可集成新颖的放置策略、负载均衡策略等。

- 我们为 MicroFog 框架的主要组件提出了部署架构，以确保它们能够在联邦雾与云环境中实现可扩展且容错的部署。

- 我们创建了该框架的概念验证原型，并通过多个用例和集成于控制引擎的基准策略展示和评估了框架的主要功能。

本文其余部分安排如下。在第2节中，我们将全面介绍基于微服务的应用放置背景，据此推导框架的需求，并分析相关研究。第3节介绍 MicroFog 框架，第4节详细说明框架主要组件的部署架构。第5节介绍用于访问 MicroFog-CE 的 API。第6节评估框架的功能。最后，第7 节总结全文。

2. 背景与相关工作

本节将全面介绍雾计算范式、基于微服务的应用及其部署相关方面，以及雾应用放置问题，以推导联邦雾环境中基于微服务的物联网应用可扩展部署框架的需求。此外，我们还对现有框架进行定性比较，以突出所提出框架的能力。

2.1. 雾计算

雾计算在物联网设备与云之间引入了一个中间层，该层由分布式、异构且资源受限的资源构成，与云数据中心相比资源受限程度更高 [2]。随着物联网应用的快速增长，雾计算正朝着联邦式多雾多云架构演进 [5] , 在这种架构中，多个雾服务提供商在雾层内提供包括计算、存储和网络资源在内的基础设施。这有助于克服雾设备资源受限的特性，实现泛在访问，并支持具备位置感知能力的应用程序放置。在这项工作中，我们考虑由不同服务提供商提供多个此类雾集群的情况；这些服务提供商与相邻集群及云保持连接。

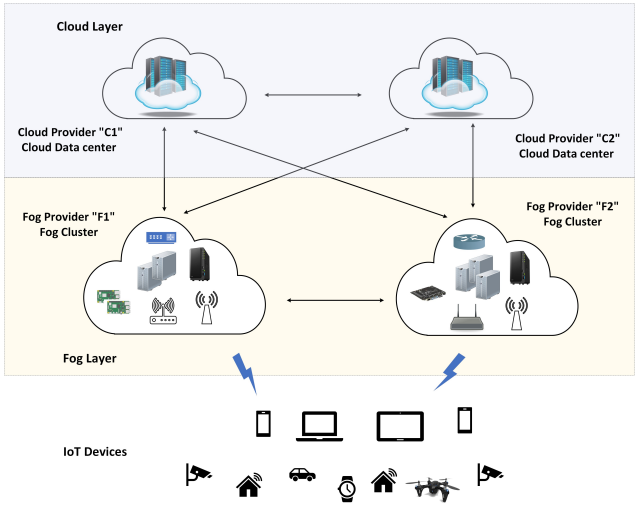

图 1： 联邦式多雾多云架构

2.2. 基于微服务的应用

微服务架构（MSA）将应用程序分解为一组可独立部署的模块，称为微服务。这些微服务围绕业务逻辑设计，具有明确的业务边界 [18]。微服务通过轻量级应用程序接口相互通信，以创建供最终用户访问的复合服务。

这些微服务具有松耦合特性，因此能够在分布式环境中独立部署和扩展。因此，动态服务发现和负载均衡机制可确保微服务之间的连接顺畅。为实现这种云原生行为，微服务被打包为容器（即 Docker），能够根据请求需求快速扩展或缩减。借助这些技术，微服务架构（MSA）可以在分布式多雾多云环境中部署微服务，同时确保水平扩展实例之间连接顺畅并实现动态负载均衡。

2.2.1. 微服务应用建模

由于基于微服务的应用中各微服务之间存在交互，因此可以使用有向无环图（DAG） [10]对其进行建模，其中 DAG 的顶点表示微服务（ $m \in M_a$ 其中 M_a 是应用程序 a中的微服务集合）。DAG 中的有向边表示微服务调用，方向从客户端微服务（消费者）指向被调用微服务（被消费方）。微服务独立打包，且资源需求各不相同，可根据满足特定请求速率/吞吐量所需的 RAM、CPU、存储等来定义。由于微服务粒度较细，它们通过相互通信来构建组合服务，其中每个应用都提供多个服务（S_a：应用程序 a的服务集合），且这些服务具有不同的服务质量要求，可在服务级别进行定义。由于微服务可能采用复杂的交互模式来构建组合服务（即，链式、聚合器、混合式），微服务之间的数据流可以是单向或双向的（df^a： $m \in M_a$之间的数据流集合）。因此，每个应用都可以表示为一个元组 $< M_a,\ df^a,\ S_a >$，其中每个服务 $s \in S_a$ 都用一个元组表示，该元组包含其微服务、其中的数据路径以及该服务的服务质量要求；$< M_a^s,\ P_a^s,\ Req_a >$。数据路径是组合服务中数据流的集合，可用于计算该服务的总完成时间。这取决于组合服务中微服务的交互模式（即，链式模式只有一条数据路径，而聚合器会调用多条数据路径）。

2.3. 应用部署相关方面

微服务应用程序的部署和管理得益于三种云原生技术：容器化平台（即 Docker）、容器编排系统（即 Kubernetes、Docker Swarm）和服务网格平台（即 Istio、Consul）。本文提出的 MicroFog 框架使用 Docker、Kubernetes 和 Istio 来部署和管理微服务。因此，下面将介绍各项技术及其与应用联邦雾云部署相关的特性：

2.4. 使用 Docker 进行容器化

微服务被打包为 Docker 容器，使其独立于主机环境。此外，与早期使用的虚拟机相比，容器更加轻量，启动时间也更短。因此，微服务容器化适用于在异构且资源受限的雾节点上进行分布式部署和扩展。Docker 容器镜像通过容器注册表存储和分发。Docker 提供了一个完全托管的容器仓库，名为 DockerHub。不过，这是一个集中式仓库，在隐私和安全方面存在局限。从集中式仓库拉取镜像可能会增加雾计算环境中微服务部署的延迟。因此，对于雾计算而言，考虑到雾基础设施托管注册表的资源可用性，探索分布式容器镜像注册表十分重要。

2.4.1. Kubernetes 作为容器编排平台

按照微服务架构分解应用程序，会产生大量微服务；为了满足吞吐量需求而对微服务实例进行水平扩展部署、为确保可靠性而冗余部署微服务实例、以及在雾计算集群中进行分布式放置以支持位置感知等因素，又会产生更多数量的容器。因此，需要使用 Kubernetes 等容器管理平台来管理成千上万个容器的生命周期。作为最受欢迎的开源容器编排器之一，Kubernetes 通过 k3s 等发行版迅速改进，以适用于异构计算环境。k3s 是一种面向极端边缘（即资源受限的物联网设备、树莓派等）的精简 Kubernetes 发行版。因此，

云服务提供商和电信服务提供商为将类云服务扩展至网络边缘，正在大力探索在多雾、多云环境中使用 k8s 和 k3s [19, 20, 21]。因此，我们总结了 Kubernetes 中使用的基本概念。为了大规模部署容器并维持微服务容器之间的通信，Kubernetes 提供了内置的**“资源”**（即 Pod、Service 等），为底层管理操作提供抽象。下面将讨论框架中一些最常用的资源。

• Pod：Pod 是 Kubernetes 支持的最小可部署单元，每个 Pod 可包含一个或多个容器（与其边车容器共同部署的容器）。Pod 代表一个逻辑主机，其中所有共同部署的容器共享网络资源，并通过本地主机进行通信。通过向 Pod 添加节点选择约束（即节点选择器、节点名称等），可将 Pod 部署到特定节点，从而对微服务实例部署进行精细控制。

• Service：Kubernetes 服务是 Kubernetes 集群中一组 Pod 的抽象，为这些 Pod 提供发现和负载均衡功能，从而允许动态创建和销毁 Pod。虽然集群内的服务发现由服务处理，但仅凭 Kubernetes 无法实现多集群服务发现。

• Namespace：命名空间可隔离具有命名空间的 Kubernetes 对象（即 Pod、服务等），从而为多租户 Kubernetes 集群中的资源隔离提供一种方式。

• ConfigMap：ConfigMap 以键值对的形式存储配置，从而将配置与 Pod 分离。这提高了容器化微服务的灵活性和可移植性。

• Secret：Secret 与 ConfigMap 类似，但用于存储不应保存在应用程序代码中的敏感信息。

• 角色和角色绑定：它们授予对 Kubernetes 资源（即节点、Pod、ConfigMap 等）的基于角色的访问权限。

2.4.2. Istio 作为服务网格

虽然 Kubernetes 提供了容器编排所需的基本功能，但在微服务应用的服务发现、负载均衡、可观测性、容错和安全管理方面仍存在局限性。因此，引入服务网格作为 Kubernetes 之上的软件抽象层，以克服这些局限。为此，Istio 实现了多个自定义资源定义（CRD），扩展 Kubernetes 资源定义，具体如下：

• 虚拟服务 (VS)：虚拟服务提供了一种为通过 Kubernetes 服务公开的 Pod 定义流量路由规则的方式，从而更好地控制流量路由。

• 目标规则 (DR)：应用虚拟服务路由规则并将流量路由到目标后，会应用目标规则来执行负载均衡、将流量定向到服务子集等操作。

• 网关：网关是集群入口流量和出口流量的负载均衡器抽象。此外，为支持分布在不同网络中的 Kubernetes 集群之间的集群间流量，Istio 提供了一种专用网关，即东西向网关。

Kubernetes 和 Istio 提供 HTTP REST API，用于检索、创建、更新和删除上述资源。此外，还提供客户端库（如 Fabric8、client-go 等），可通过编程语言访问这些 API。

2.4.3. 应用部署示例

本节将演示如何使用 Kubernetes 和 Istio 资源，在可用的 Kubernetes 和 Istio 集群中部署基于微服务的物联网应用。我们以智能健康监测应用（见图 2）[10] 为例。该应用包含三个微服务和两个供用户访问的组合服务：一个时延敏感型紧急事件检测服务（$S1$），其两个微服务（$m1$，$m1$）均部署在分布式雾资源上；另一个是由两个微服务（$m1$，$m3$）组成的时延容忍型预测健康预警服务。$m1$ 由这两项服务共享，并部署在雾层中，以满足服务 $S1$的严格延迟需求，而 $m3$ 则部署在云中。

图 2 展示了 Kubernetes 和 Istio 资源的逻辑视图，说明它们如何将来自用户的外部流量路由至 $m1$ 和$m3$，并维持互连微服务之间的通信（在 $m1$ 和 $m2$之间，在 $m1$ 和 $m3$之间）。

2.4.4. Kubernetes + Istio 多集群支持

Istio 支持将单个网格部署到多个 Kubernetes 集群中，从而实现跨集群服务发现和负载均衡。面向多集群场景的 Istio 部署模型取决于底层网络模型的性质。最简单的网络模型是多个集群属于同一个网络，所有节点都通过 VPN 等技术完全互连。然而，跨越多个 Kubernetes 集群的大规模生产系统往往属于多个网络，这些网络具有管理边界，且每个集群都通过负载均衡器对外提供服务。本文所考虑的雾计算架构（第 2.1节）对应于多网络模型。因此，本文采用具有多个控制平面的 Istio 多网络部署，以提高部署的弹性。在这种部署模式下，每个 Istio 控制平面都会连接到所连接集群的 API 服务器，以实现跨集群服务发现。

Istio 引入了一个东西向网关，将集群内的服务暴露给其他集群，以实现跨集群服务发现。此外，为确保跨集群 DNS 查找成功，消费者集群需要

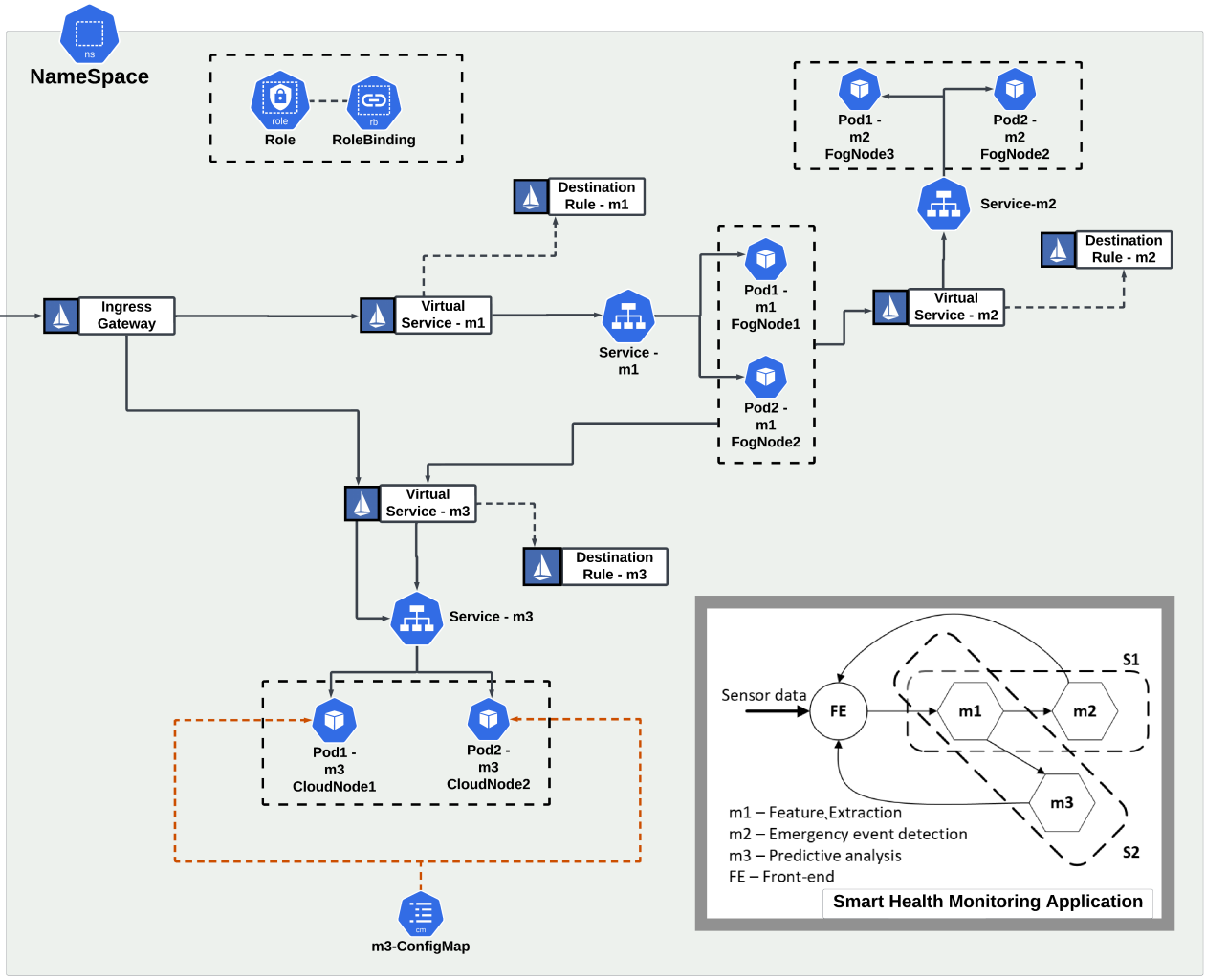

图 2： 智能健康监测应用程序的部署示例

能够访问部署在其他集群中的被消费方的 Kubernetes Service 资源、Istion 目的地规则（DR）和虚拟服务（VS）。例如，对于 $S1$一个示例应用程序，要将流量从其雾计算集群路由到部署在云集群中的 $m2$，与 $m3$ 相关的上述资源都应部署在雾计算集群和云集群中。

2.5. 放置问题

雾计算环境中的基于微服务的物联网应用放置问题，涉及在联合雾计算与云计算环境中部署和维护微服务，以满足应用服务的服务级别协议 (SLA) [8, 22]。

得益于微服务架构所提供的灵活性，放置算法旨在纳入水平可伸缩性以满足吞吐量要求 [8, 9, 10],位置感知分布 [11], 冗余部署以提升可靠性 [23], 根据服务发现能力，在雾集群和云之间实现均衡放置[24, 10], 最优负载均衡和路由 [25], 等，从而在满足服务质量的同时，高效利用有限的雾资源参数，例如总完成时间、预算、可靠性、可用性和吞吐量。

放置算法可以通过批量部署的方式执行 [26, 10], 一次处理多个应用部署请求；也可以通过顺序部署的方式执行 [27, 24]，依次处理队列中的部署请求。此外，放置策略可以设计为集中式 [28] 或分布式 [29] 算法，以便在多个基础设施提供商提供的分布式雾计算与云计算资源中进行放置。

2.6. 框架要求

基于上述背景，我们总结了在联邦雾计算与云计算环境中实现基于微服务的物联网应用可扩展部署的框架所需的功能需求和非功能需求，如下所示：

- 多雾多云微服务放置和部署：框架应支持使用集中式或分布式运行模式，在多个雾和云集群中执行放置算法。相应地，

表 1现有框架对比

Work	架构		云原生应用支持					微服务组合支持					控制引擎			数据存储
Work	集成	多集群	μ服务	容器	容器编排	服务网格	自动化部署	服务发现		负载均衡			可扩展性	可伸缩性	可配置性	
								可用性	跨集群	可用性	可配置的	跨集群				
[30]	雾、云	-	✓	✓	-	-	∂	∂	-		-	-	✓	∂	∂	集中式
[17]	雾、云	-	✓	✓	✓	-	∂	✓(Kubernetes)	-	✓(Kubernetes)	-	-	✓	∂	∂	分布式
[31]	雾、云	-	✓	✓	✓	-	∂	✓(Kubernetes)	-	✓(Kubernetes)	-	-	✓	∂	∂	-
[32]	Edge	-	✓	✓	✓	-	-	✓	-	✓	-	-	✓	∂	∂	-
[33]	雾，云	-	✓	✓	✓	-	∂	✓(Docker Swarm)	-	-	-	-	∂	∂	∂	-
[34]	雾、云	-	-	-	-	-	∂	-	-	-	-	-	∂	∂	∂	分布式
[16, 35]	雾、云	-	∂	✓	∂	-	∂	∂ (代理服务器)	-	-	-	-	∂	∂	∂	分布式
[36]	雾、云	-	✓	✓	-	-	∂	∂	-	-	-	-	✓	∂	∂	集中式
Our	雾、云	✓	✓	✓	✓	✓	✓	✓ (Kubernetes、Istio)	✓	✓ (Istio)	✓	✓	✓	✓	✓	分布式、复制容错

✓：框架支持， ∂：部分支持

应用微服务需要使用相关的 Kubernetes 和 Istio 资源进行部署。

• 在混合式环境中无缝组合微服务：Kubernetes 和 Istio 资源部署应确保跨集群服务发现和负载均衡。

• 能够轻松集成新颖的放置算法和负载均衡策略。

• 支持异构云原生应用程序部署，无需进行任何应用级更改。

• 兼容云原生技术，使框架能够随着底层技术的发展而改进和演进（可扩展性）。

• 可配置控制引擎，支持不同的运行模式，例如集中式或分布式运行；支持不同的应用放置模式，例如事件驱动或周期性放置请求处理，以及批量或顺序处理放置请求。

• 分布式存储解决方案，用于存储应用放置和部署所需的数据（即应用模型、Kubernetes 和 Istio 资源定义）。

• 快速原型开发支持，使评估放置 - 算法能够融入其快速设计和开发周期。

• 框架应具备灵活性和可扩展性，以便部署并运行于分布式雾和云集群中。

2.7. 现有雾计算框架

在本节中，我们根据上一节确定的需求，对现有雾计算框架进行定性比较（见表 1）。

Yousefpour 等人 [30]提出了 FogPlan，这是一个利用 Kubernetes 或 Open-Stack 等容器编排平台动态配置容器化雾服务的框架。FogPlan 由一个集中式雾服务控制器组成，负责托管数据存储、配置雾服务，并将其部署到雾节点中。Santoro 等人 [17] 提供了一个名为 Foggy 的、基于开源技术（即 OpenStack、Kubernetes、Docker）的平台，用于在雾计算环境中放置工作负载。FogAt-las [31]通过扩展 Kubernetes 来扩展 Foggy 平台以便以用户友好的方式编排分布式雾计算与云计算资源。Ermolenko 等人 [32] 还提出了一个基于 Kubernetes 和 Docker 的框架，在移动边缘计算（MEC）环境中部署 Kubernetes 集群。Bellavista 等人 [33] 创建了一个基于 Docker 和 Docker Swarm 的微服务部署框架，并在云端部署了一个集中式控制引擎，用于执行放置算法并据此部署微服务。尽管他们利用 Kubernetes 和 Docker Swarm 的功能进行容器编排，但在多集群支持、借助服务网格技术实现高级微服务组合，以及控制引擎在多雾多云环境中的可伸缩性方面仍存在局限。Tuli 等人 [34] 介绍了 FogBus 框架，用于利用边缘/雾计算资源和远程云资源，放置由互连模块构成的应用程序。Deng 等人 [16] 提出了 FogBus2，这是一个资源管理框架，用于在边缘和云资源之间部署容器化应用程序，这些资源通过 VPN 网络相互连接。Wang 等人 [35] 改进了 FogBus2，并通过 Kubernetes 将容器编排能力集成到该框架中。他们的框架支持集成新型放置策略并对其进行性能监控，以评估这些新型放置策略。然而，该框架不支持由多个地理分布式 Kubernetes 集群构成的多集群场景。此外，由于服务发现和负载均衡方面的局限，该框架不支持微服务的动态组合，也未集成服务网格技术，无法充分发挥微服务架构的能力。FogBus2 中 Kubernetes 资源的使用仅限于 Pod，这限制了该框架的可伸缩性。此外，要在该框架中部署容器化应用程序模块，还需要进行应用级更改。Mahmud 等人 [36] 提出了一个名为 Con-Pi 的完全分布式、可伸缩框架，用于执行基于微服务的应用。Con-Pi 提供了一个集中式控制器，用于执行集成的定制化放置策略，并据此部署容器化微服务。然而，Con-Pi 不支持高级微服务组合、已部署微服务的动态服务发现和负载均衡，也未考虑跨多个雾资源集群的应用部署。

根据表 1 所提供的定性分析，现有框架在多项需求方面存在局限

，这些需求在第 2.6 节中有所说明，例如多雾多云放置和应用程序的全自动部署；确保通过容器编排器和服务网格技术实现跨集群的微服务动态组合；通过开源技术提高框架的可扩展性；确保框架在高度分布式雾计算环境中的可伸缩性；提供可配置性以支持不同的运行和放置模式；以及对应用放置和部署所需数据进行分布式管理。因此，本文提出了一种新颖的框架，用于在满足上述需求的联邦雾环境中进行基于微服务的应用放置。

3. MicroFog 框架

本节将介绍所提出的 MicroFog 框架的高层架构、主要组件和工作流程，以说明 MicroFog 如何满足第 2.6 节中确定的需求。

3.1. 高层架构

图 3 展示了 MicroFog 的高层架构和工作流程。MicroFog 提供了一个可扩展且易于扩展的控制引擎（CE），用于执行放置算法，并在安装了 Istio 的 Kubernetes 集群中部署物联网应用。CE 与三个数据存储进行通信：1. YAML 文件存储库，其中包含部署应用所需的 YAML 定义（包括 Kubernetes 和 Istio）；2. 元数据存储，用于存储应用模型，以及指向 YAML 文件存储库中相关部署资源的链接；3. Docker 镜像仓库，用于托管应用微服务的 Docker 镜像。应用提供商可以向 MicroFog-CE 提交部署请求（PR），定义待部署的应用及其服务质量要求。CE 接收应用部署请求（PR），根据选定的放置策略（内置放置算法或通过 API 调用的外部算法）进行处理，按照生成的部署位置和负载均衡策略配置相关的 Kubernetes 和 Istio YAML 文件，最后使用 Kubernetes API 在雾计算与云计算资源中部署这些应用。此外，MicroFog 集成了监控和日志工具，用于观察 MicroFog 框架及通过该框架部署的应用的性能。

3.2. 主要组件和技术*3.2.1.* 控制引擎（*CE*）

CE 旨在抽象微服务放置（执行放置算法和部署）以及跨领域功能处理（即服务发现、负载均衡），以支持在多雾多云环境中动态组合微服务。

我们将 CE 实现为一种可独立部署和扩展的微服务，使用 Quarkus [1], 开发。Quarkus 是一种新颖的 Kubernetes 原生轻量级 Java 框架，旨在构建云原生微服务。Quarkus 可减少内存使用量并提高部署密度 [37], ，因此适用于开发将在资源受限的雾计算环境中部署的微服务。由于 Quarkus 是 Kubernetes 原生框架，CE 的开发和部署变得简单直接、耗时更少，从而使用户能够根据不断变化的需求快速改进、扩展和定制它。因此，Quarkus 正迅速成为用于创建云原生微服务的热门轻量级 Java 框架。此外，Quarkus 通过其扩展 [2] ，可轻松使用 Fabric8 Kubernetes 和 Istio 客户端。Fabric8 是广受欢迎的 Kubernetes 和 Istio 客户端，可全面访问 Kubernetes API。Fabric8 提供了丰富的领域特定语言（DSL），用于与 Kubernetes API 交互，因此成为最常用的开源 Kubernetes 客户端之一，拥有一个极其活跃、持续使用并改进它的社区。因此，我们选择了 Quarkus 与 Fabric8 Kubernetes 和 Istio 客户端来创建控制器。

我们将对 MicroFog-CE 的功能特性和非功能特性进行如下讨论：

1. 放置的 *PR* 提交： 应用提供商可以通过 API 向 CE 提交 PR，该 API 接收包含 JSON 格式 PR 的 HTTP POST请求（图中所示的 API 1）。每个提交的 PR 都可以定义与应用程序相关的多个数据字段，包括应用 ID、QoS参数、应用程序放置的任何限制条件、流量入口集群等。提交后，CE 会利用这些信息处理 PR（即，应用ID 用于从数据存储中检索应用模型和部署资源，入口集群表示充当相关应用入口流量入口点的集群），并据此部署应用微服务和部署资源。

2. 多种运行与放置模式： CE 支持集中式和分布式运行模式。在集中式模式下，主 CE（即部署在云中的 CE）拥有基础设施的全局视图（即雾、云集群、它们的拓扑和资源可用性），负责执行放置算法。在此模式下，主 CE 会查询次级 CE（通过 API 3），获取每个集群内可用资源的信息以及与拓扑相关的数据（即从每个集群可直接访问的雾和云集群），以构建联邦环境的全局视图。主 CE 利用这些信息，为 PR 请求的应用生成放置方案，并将放置结果详情发送到各个相关集群（通过 API 4）。部署在各集群中的次级 CE 会处理放置结果，并相应地部署 Kubernetes 和 Istio 资源。相比之下，在分布式模式下，所有 CE 都负责在各自集群本地运行放置算法。它们通过在集群之间转发 PR，协同进行

[1] https://quarkus.io/

[2] https://github.com/fabric8io/kubernetes-client

MicroFog 框架

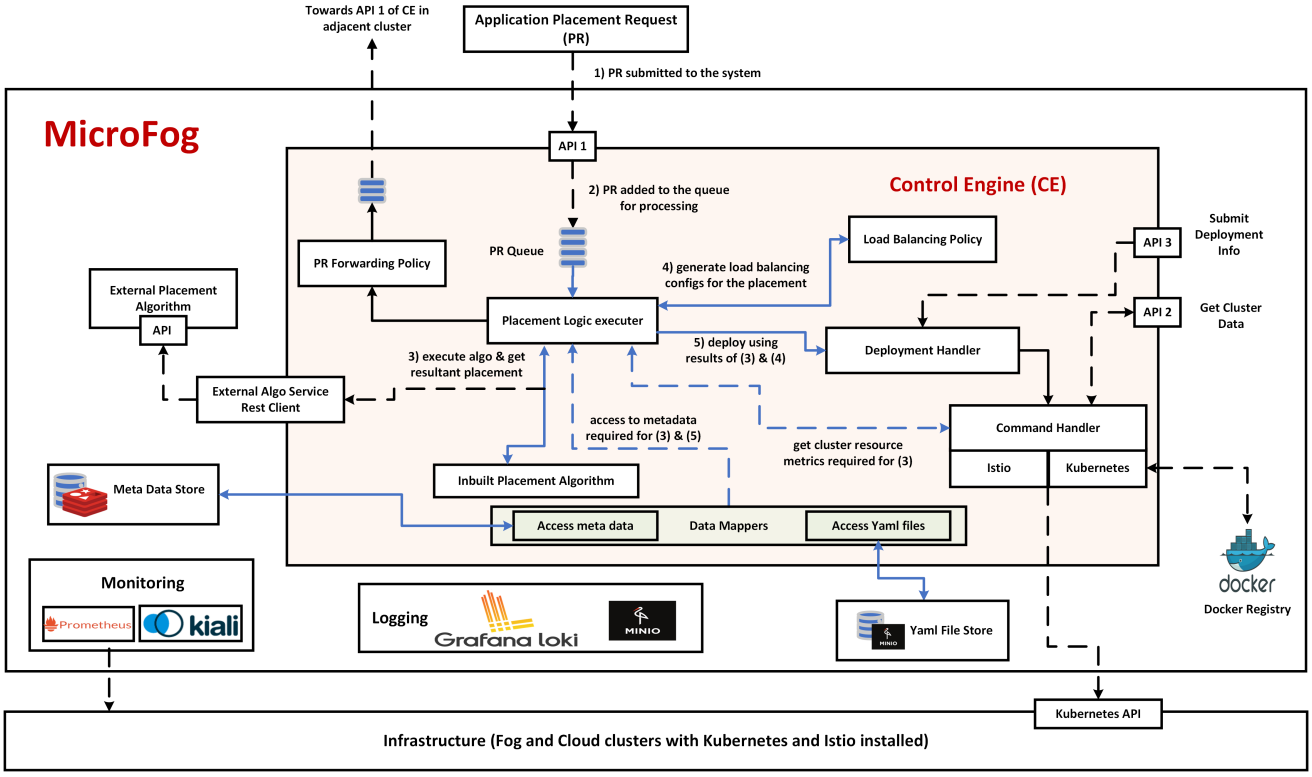

图 3： MicroFog：高层架构

多雾多云环境中的分布式放置。MicroFog-CE 使用 API 1 在集群之间转发 PR。

此外，控制引擎支持两种放置模式：周期性放置和事件驱动放置。周期性放置会根据可配置的时间间隔定期调用放置算法。在此模式下，放置算法可以设计为批量处理 PR（队列中的所有 PR 由算法同时处理），也可以按顺序处理（按照先进先出顺序处理或按优先级处理）。在事件驱动模式下，收到新的 PR 后会调用放置算法。

3. 放置算法集成： 控制引擎支持轻松集成新型放置算法。可通过两种方法实现：一种是在控制引擎中内置算法，即扩展控制引擎的*PlacementAlgorithm.java* 基类来实现新的放置策略。该基类会初始化放置算法所需的元数据（即设备的资源可用性、应用模型和拓扑信息）。新型放置算法可扩展该基类，实现自定义放置逻辑，利用元数据生成放置结果（由 PlacementOutput.java 表示），其中包括微服务到设备的映射和 PR 完成数据（已完成的 PR 与未完成的 PR；未完成的 PR 应通过转发流程发送到其他集群，以完成放置）。此外，

控制引擎还支持集成外部放置算法，因此算法可以使用其他编程语言实现（例如，使用机器学习的放置算法可用 Python 实现）。此类算法可以作为独立微服务实现，并通过实现一个 API 将其集成到 MicroFog-CE 中；该 API 可由控制引擎的 图中外部算法服务 *REST* 客户端调用 *3* ，并通过 HTTP GET 请求发起调用。控制引擎的 REST 客户端会随 GET 请求发送元数据，以便外部放置算法生成放置结果，并将部署相关信息返回给控制引擎。

默认情况下，MicroFog-CE 实现了 [29] 中提出的延迟感知可扩展放置策略。该算法旨在将延迟关键型服务的微服务尽可能靠近访问它们的用户放置。我们以分布式和集中式两种模式实现了该算法。此外，我们还实现了微服务具备和不具备水平可扩展性的两种版本，以展示 MSA 在资源受限的雾计算环境中能够带来的性能提升。

4. 访问基础设施指标： 为了做出放置决策，放置算法需要与基础设施相关的指标，例如集群内的资源可用性。为此，当前版本的控制引擎提供了两种测量方式：1. 控制引擎访问 Kubernetes 指标服务器，以获取当前 CPU 和 RAM 使用情况的节点指标；2. 控制引擎还通过查询 Kubernetes API，提供当前已部署 Pod 的资源分配情况。

放置算法可以利用这两类指标信息来做出放置决策。指标收集还可以进一步扩展，以使用 Prometheus，从而利用时间序列指标数据来制定放置决策。

5. 负载均衡策略集成：由于微服务具有可独立部署和扩展的特性，负载均衡对于在联邦雾计算与云计算环境中跨水平扩展的微服务合理分配负载至关重要。默认情况下，Istio 使用轮询负载均衡方法来路由请求。此外，Istio 还支持随机、最少请求和加权负载均衡等其他方法，这些方法已在 Istio 用于服务发现和负载均衡的 Envoy Proxy 中实现。通过更新与各微服务相关的 Istio 目标规则，即可配置这些方法。除此之外，MicroFog-CE 还提供增强功能，以支持自定义负载均衡策略，并可根据自定义负载均衡策略更新加权负载均衡方法中的权重。

例如，当前版本的控制引擎实现了加权轮询负载均衡策略。根据放置情况计算出每个微服务实例的权重后，控制引擎会处理 Istio 虚拟服务和目标规则资源中与子集、权重和路由相关的更新。对于集中式运行模式，这种更新很直接；而分布式放置则面临一个主要挑战：只有在所有所需的微服务实例都完成放置后，才能计算负载均衡信息。此外，为了正确执行负载均衡策略，Istio 需要在托管特定微服务（被消费方微服务）的所有集群中，以及任何尝试与其交互的微服务（消费者微服务）所在的集群中，都具备虚拟服务和目标规则资源。因此，在分布式放置模式下，对于每个微服务，控制引擎都会等待其所有实例及其消费者微服务完成放置。随后，通过分布式控制引擎的 API 3，将虚拟服务和目标规则更新所需的信息（子集名称和权重）发送到相关集群。

6. *PR* 转发策略集成：在多云多雾计算环境中进行放置，需要在多个云计算和雾计算基础设施提供商提供的基础设施上使用分布式放置策略。MicroFog-CE 支持更新部分处理后的 PR 状态，并将其转发至相邻的雾计算集群或云集群。此类 PR 会提交至所选集群的 API 1。此外，也可以集成新颖的转发策略。CE 的默认实现提供了两种转发策略，可将 PR 转发至随机雾计算集群或云端。由于各 CE 实例独立配置，因此可以在不同集群中使用不同的转发策略。

7. 自动化应用部署：MicroFog CE 抽象化了面向框架用户的微服务部署过程。对于每个应用程序，YAML 文件存储库用于检索与微服务部署相关的 Kubernetes 和 Istio 资源。这包括不同抽象级别的资源，例如：1. 应用程序级资源，如命名空间、角色和角色绑定；2. 微服务级资源，如 ConfigMap、Secret 和 Pod 定义 YAML 文件，用于根据放置算法的输出，在映射的节点上创建微服务实例；3. Service、虚拟服务和目标规则，用于跨集群的服务发现，以及根据负载均衡策略对流量进行负载均衡和路由，从而创建复合服务；4. 网关，用于使入口流量能够到达应用有向无环图的根微服务。此外，MicroFog-CE 通过在相关集群中部署微服务组合相关资源（即 Kubernetes 服务、虚拟服务和目标规则），实现跨多个雾和云集群的联邦。CE 规则用于处理这些功能，从而将底层复杂性从框架用户中抽象出来。

8. 可扩展的分布式 *CE* 部署：由于 CE 是基于 Kubernetes 原生微服务框架开发的微服务，因此可以在启用了 Kubernetes 和 Istio 的环境中以分布式方式部署。每个 CE 都可以单独配置，并通过 REST API 跨集群通信，从而使 MicroFog 能够扩展至联合雾计算和云环境中运行。

9. 可扩展性：CE 的设计与架构通过将 Java 对象实现为丰富的领域专用对象，捕捉了基于微服务的应用放置问题域。图 4 展示了用于开发 MicroFog-CE 的领域图，该图遵循第 2 节中制定的系统模型和放置问题。这使 CE 的实现易于理解和扩展，以便纳入新功能。此外，由于 MicroFog 框架与开源云原生技术兼容，CE 可以随着底层技术能力的发展而不断演进。

10. 可配置性：Quarkus 支持通过 Kubernetes ConfigMaps 获取应用程序配置属性。这大大提高了 CE 的可配置性，用户无需创建新的 Docker 镜像，即可更新应用程序配置，从而快速采用不同的配置（策略、放置模式、运行模式等）。

3.2.2. 数据存储

MicroFog 使用三种主要的数据存储，如下所示s:

1. 元数据存储：元数据存储包含应用程序相关信息，分为两大类：1）应用模型（如第 2.2.1 节所述），其中包含与微服务相关的规范，以及微服务之间相互连接以创建服务、数据流的相关信息，

等等。2）与应用部署相关的 Kubernetes 和 Istio 资源。这包括资源类型（即命名空间、Pod、服务等）以及包含每个资源规范的 YAML 文件 URL。我们使用 [3] Redis 作为主数据库来存储这些信息。尽管 Redis 最初是作为缓存推出的，但如今它越来越多地被用作主数据库，以降低数据检索的复杂性并提升性能。Redis 支持以键值对的形式存储数据。借助 Redis Java 客户端 Redisson， *Application*控制引擎的领域对象可以轻松序列化并存储在 Redis 元数据存储中，也可以将其取回为 Java 对象。

2. *YAML* 文件存储库： 该存储库用于以 YAML 文件的形式存储 Kubernetes 和 Istio 资源配置。由于雾集群具有地理分布式特性，因此需要分布式对象存储来高效存储 YAML 文件。为满足这一要求，我们使用 MinIO 对象存储 [4], 这是一种兼容 AWS S3、原生支持 Kubernetes、专为多雾多云环境设计的对象存储。对于每个要部署的 Istion/Kubernetes 资源，CE 会使用对象 URL 从 MinIO 数据存储中检索 YAML 文件，并使用 Fabric8 Kubernetes 客户端库将其加载为表示部署资源的领域对象。

3. *Docker* 镜像仓库： 由于物联网应用微服务经过容器化后才能部署，因此必须将容器镜像存储在 CE 可访问的 Docker 镜像仓库中。在当前实现中，我们使用 Docker Hub，这是一个公开可用的托管式 Docker 存储库。不过，根据各雾集群托管镜像的资源可用性，还可以进一步改进：将本地 Docker 存储库与 Docker Hub 配合使用。

3.2.3. 监控与日志管理：

由于云原生微服务具有高度分布式且动态的特性，监控和可观测性仍是其中至关重要的方面。为此，Istio 支持以预配置插件的形式集成多种工具，包括指标收集与可视化（Prometheus 和 Grafana）、分布式追踪（Jaeger、Zipkin），以及使用 Kiali 实现服务网格可视化。在当前版本的 MicroFog 框架中，我们集成了 Prometheus、Kiali 和 Grafana，以观测跨集群流量并验证 MicroFog-CE 的功能。此外，MicroFog 使用集群级日志架构来管理各集群内生成的日志。为此，MicroFog 使用 Grafana Loki，这是一种去中心化、轻量级的日志栈，可压缩数据并将其存储在 S3 等对象存储中。由于用于存储 YAML 文件的 MinIO 对象存储兼容 S3，MicroFog 使用同一存储来保存日志。与 ElasticSearch 等其他云原生日志解决方案相比，Loki 的架构较为简单，所需存储空间更少、功耗更低，因此适用于雾计算部署。根据雾集群的资源可用情况，可以将日志存储在云端托管的 MinIO 中，以节省存储空间。不过，也可以根据需求轻松集成其他工具。此外，当前架构可以轻松扩展，使 MicroFog-CE 能够利用监控和日志工具收集的指标来执行动态放置算法，或集成基于机器学习的方法。

3.2.4. 快速原型开发支持

新型放置算法的开发需要经历多个开发和评估周期，以优化其性能。因此，在进行大规模评估或将其应用于真实世界的应用部署之前，在策略开发的不同阶段进行快速原型开发十分有益。得益于对开源云原生工具的使用，MicroFog 可以利用 Kind 和 MetaLB 等工具快速创建底层基础设施，从而创建由异构节点组成的雾计算集群，并通过负载均衡器路由集群间流量。

3.3. MicroFog-CE 的拉取请求处理流程

本节将讨论 MicroFog-CE 关于处理所接收拉取请求的高级伪代码（见算法 1）。在每个集群都包含一个独立 CE 的环境中，当采用分布式放置模式时，所示拉取请求处理流程会在所有 CE 中执行；当放置模式设置为集中式放置时，则仅在主 CE 中执行。

拉取请求处理首先从*PR*队列*PR*队列中获取拉取请求（第 1 行）。获取方式取决于控制引擎的放置模式：在周期性放置模式下，会取出*PRQueue* 中的所有拉取请求进行处理；而在事件驱动模式下，每添加一个拉取请求，就会立即从队列中取出。如果拉取请求处理线程正忙，该拉取请求会留在队列中，直到线程空闲。当前控制引擎的实现使用单个线程处理拉取请求，而多个线程则将传入的请求添加到 *PRQueue* 中。该队列使用*Java ConcurrentLinkedQueue*实现，是一种非阻塞、线程安全的队列实现。

检索到的 PR 分三个主要步骤处理：元数据检索、放置算法执行，以及最后使用 Kubernetes 和 Istio 资源部署基于微服务的应用，并处理未完成的 PR。元数据检索的第一步是生成放置算法所需的集群数据（第 5–11 行）。这包括集群中每个节点的资源可用性详情，以及拓扑详情，例如各个待考察集群相邻的雾和云集群。对于集中式放置，负责执行放置算法的主 CE 需要全面掌握所有雾和云集群的情况。因此，主 CE 会向已连接集群的 API 2 发送请求，以查询其他集群（第 10–11 行）。为此，我们实现了一个响应式 REST 客户端，它会同时发送所有请求，等待所有已发送请求的结果，并从响应中检索各集群的数据。Quarkus 框架支持的响应式 REST 客户端

[3] https://redis.io/ [4] https://min.io/

支持并发发送请求，从而提高从分布式集群收集数据的效率。作为元数据检索的第二步，CE 会从 Redis 元数据存储中查询与每个 PR 所请求应用程序相关的应用模型（第 13 行）。由此检索到一个类型为 *Application* 的 Java 领域对象（如领域模型4所示），该对象包含微服务、复合服务、数据路径、数据流、资源需求以及用于微服务部署的命令；这些内容均以可序列化 Java 对象表示。

随后，CE 开始使用放置算法处理拉取请求s（第 16–19 行）。由于 CE 支持通过扩展现有 CE 或将放置算法作为外部微服务来集成算法，因此可以将算法配置为 CE 的一个属性。CE 采用工厂模式，根据配置的放置算法名称初始化放置算法。因此，要在内部集成放置算法，必须将其添加到工厂中。若要使用外部算法，CE 会实现一个 REST 客户端，并提供可配置的 URL，以便将其更新为外部算法的 URL（第 19 行）。

放置算法生成放置结果后，CE 随即进入应用部署阶段。在此步骤中，CE 会为每个集群生成两大类部署信息：基本部署信息和负载均衡信息。基本部署信息包括 Pod 到设备的映射及所需的资源分配、每个应用程序的入口集群（用于部署 Istio 网关和相关的虚拟服务，以实现入口流量路由）等。与负载均衡相关的部署信息则包括执行负载均衡策略，以放置已完成的微服务，并相应地生成子集和权重。这些数据将用于更新虚拟服务和目标规则，以确保实现所需的负载均衡。

生成部署信息后，CE 会根据转发策略启用一个新线程，在分布式放置模式下转发未完成的 PR，同时当前线程继续进行部署。在集中式放置模式下，CE 使用响应式 REST 客户端并发地将部署信息发送给其他节点，同时并行执行当前集群的部署。做出这一决定是为了提高放置的整体效率，因为如果按顺序部署，以 Docker 容器形式运行的微服务可能会耗费大量时间。同样，在分布式放置模式下，与先前集群相关的负载均衡信息也会并发传输，同时一个线程继续处理与当前集群相关的部署。

4. MicroFog 部署

在联邦雾云环境中部署 MicroFog 包括两个主要步骤：1. 为数据存储进行分布式设置；2. 分布式部署 CE。作为部署场景示例，我们提供了部署架构（图5 和图 6）展示了每个步骤的示例。示例场景所用的联邦雾云环境由两个雾集群和一个云集群组成。三个集群分别属于三个独立网络，是通过 Istio 多主架构互连的三个独立 Kubernetes 集群，以支持集群间的微服务组合和流量传输。

4.1. MinIO YAML 文件存储部署

我们在图5中提供了一个示例部署场景，以展示联邦雾云环境中的 MinIO YAML 文件存储分布式部署。为了实现 YAML 文件的分布式存储和访问，我们设计的部署架构需满足以下要求：1）在集群间进行分布式部署，以降低应用部署的延迟；2）在分布式数据存储间进行复制，以保持数据一致性；3）通过优先级故障转移机制实现容错，以具备延迟感知能力的方式确保可用性。

为实现这些目标，我们使用 Kubernetes 和 Istio 资源创建了两层流量路由，即管理层和数据访问层。管理层用于配置每个集群中部署的 MinIO 服务器。管理层的 Kubernetes 服务和 Istio 虚拟服务通过入口网关公开 MinIO 的默认端口，以便访问管理控制台（控制台端口），并在分布式 MinIO 实例之间进行数据复制（API 端口）。第二层路由公开 MinIO 数据存储的 API 端口，供控制引擎访问，以获取应用部署所需的 YAML 文件。该流量层实现了两级故障转移策略，以提高部署的可靠性。Istio 支持位置感知负载均衡，可根据节点所在的区域（topology.kubernetes.io/region）、可用区（topology.kubernetes.io/zone）和子区域（topology.istio.io/subzone）执行故障转移。我们使用区域和可用区进行故障转移：所有雾级资源都属于区域“fog”，其中每个雾计算集群都视为一个独立的可用区。同样，所有云集群都属于区域“cloud”。Istio 的默认故障转移策略会优先在同一区域内进行故障转移（即雾计算集群会故障转移到相邻的雾计算集群）。我们进一步扩展此策略，引入 Istio 灾难恢复，以确保在没有可用雾计算集群时能够从雾计算环境故障转移到云环境。为确保适当的容错能力，Kubernetes 集群中的每个节点都需要标注其所属的区域和可用区。虽然当前实现仅限于两级，但也可以通过实现 Istio 子区域将其扩展到三级。

4.2. Redis 元数据存储部署

Redis 元数据流的部署采用类似的方法，包含两个流量层：一个用于数据复制，另一个用于检索应用信息。我们采用 Redis 支持的主从部署。在我们提出的架构中，我们将 Redis 主服务器部署在云集群中，其余部分则部署为副本；这些副本与主服务器同步，以获取可用的元数据。与 MinIO YAML 存储类似，此部署也利用了本地性

算法1 MicroFog-CE PR 处理

```
1: 过程处理PR(PR队列)
2:     PR ← 从PR 中获取并进行处理PR队列
3:     # 步骤1. 元数据检索由两个子步骤1.1和1.2组成
4:     # 步骤1.1：集群数据检索（包括集群内资源可用性和拓扑信息
5:     集群数据 ← {}                                          ▷将集群名称映射到其数据
6:     集群内设备数据 ←加载集群内设备数据()                    ▷已加载与当前集群相关的设备数据
7:     currentClusterData ← inclusterDeviceData ∪ topologyData
8:     clusterData.添加(currentClusterName, currentClusterData)
9:     # 对于集中式放置，使用 API 2 从其他集群请求集群数据
10:    如果 集中式放置 且 是主 CE 则
11:        clusterData ←请求其他集群数据()
12:    # 步骤 1.2：从元数据存储加载应用元数据
13:    应用信息 ←加载相关应用信息(prs)
14:    # 步骤 2：执行放置算法
15:    放置输出 ← {}
16:    如果 是内部算法则
17:        placementOutPut ← placementAlgo.生成放置(PRs, appInfo, clusterData)
18:    如果 是外部算法 则
19:        放置输出 ← externalPlacementAlo.生成放置(PRs, appInfo, clusterData, externalUrl)
20:    # 步骤 3. 使用 Istio 部署 +Kubernetes 资源并处理未完成的 PR
21:    每集群部署信息 ← {}
22:    perClusterDeploymentInfo.添加（生成基础部署信息（placementOutPut））
23:    perClusterDeploymentInfo.添加（生成负载均衡相关部署信息(placementOutPut)）
24:    如果 是分布式放置 那么                                  ▷使用独立线程
25:        incompletePRs ← placementOutPut.getIncompletePRs()
26:        forwardIncompletePRs(incompletePRs)
27:    thisClusterDeployment ← perClusterDeploymentInfo.getThisCluster()
28:    deploymentHandler.部署命令(thisClusterDeployment)
29:    发送至其他集群(perClusterDeploymentInfo − thisClusterDeployment)
```

在 Istio 中进行负载均衡，以确保从雾层向云的故障转移，从而提高数据的可用性。

4.3. 控制引擎部署

图 6 展示了一个在联邦雾云集群中分布式部署控制引擎的示例场景。我们将在下文讨论部署的主要方面：

• 控制引擎的分布式部署及集群间通信维护：在集中式和去中心化放置模式下，控制引擎都需要访问部署在不同集群中的其他控制引擎的 API，以执行各种功能，包括查询集群数据、转发 PR、提交部署信息。我们通过使用 Istio 灾难恢复和虚拟服务，根据每个请求的标头值进行路由。我们引入了一个名为“cluster”的标头，用于指定请求的目标集群。为实现正确路由，每个控制引擎 Pod 都会标记其集群名称，而目的地规则则根据集群名称创建子集。按照此实现方式，虚拟服务通过将标头值与子集标签匹配来进行路由。

• 向特定集群提交 PR：上述实现不仅支持控制引擎之间的路由，还能根据标头值将发往控制引擎的入口流量（即提交 PR）路由至特定控制引擎。

• 在部署期间分别配置每个控制引擎：为了提高控制引擎配置的效率，并使每个控制引擎能够独立配置，我们使用 Kubernetes ConfigMap 来定义控制引擎配置。由于 Quarkus 应用程序原生适用于 Kubernetes，因此该 Quarkus 应用程序被配置为从 ConfigMap 获取 application.properties 的值。

• 确保能够访问底层 Kubernetes 和 Istio 部署：控制引擎需要访问 Kubernetes API 以执行各种操作（即检索节点数据、检索资源指标、检索 Pod 数据，以及部署 Kubernetes 和 Istio 资源）。为此，应向控制引擎授予适当级别的权限。系统会创建专用服务账户，并将其关联到 ClusterRoleBinding 和 ClusterRole，以授予其在整个集群范围内所需的访问权限。

4.4. 可观测性、监控和日志工具的部署

在当前实现中，我们集成了 Prometheus 和 Kiali，以验证控制引擎支持的功能。Kiali 使用 Prometheus 监控工具创建拓扑图、计算健康状态并显示指标。Istio 插件预先配置了 Kiali，以可视化多集群服务网格，其中包括不同视图，例如图表（展示应用程序、服务、微服务版本等）、流量流向、指标详情，以及 Istio 配置（与每个已部署 Istio 资源相关的 YAML 文件）。在分布式架构中，Prometheus 和 Kiali 组件部署在每个集群中，并通过 Istio 入口网关公开 Kiali 仪表板，以便远程访问。

我们使用 Loki 和 Grafana 进行日志聚合与可视化。Loki 配置为使用 MinIO 对象存储中的对象存储桶。由于 MinIO 部署和请求路由已在（章节4.1）中处理，因此日志可以被定向

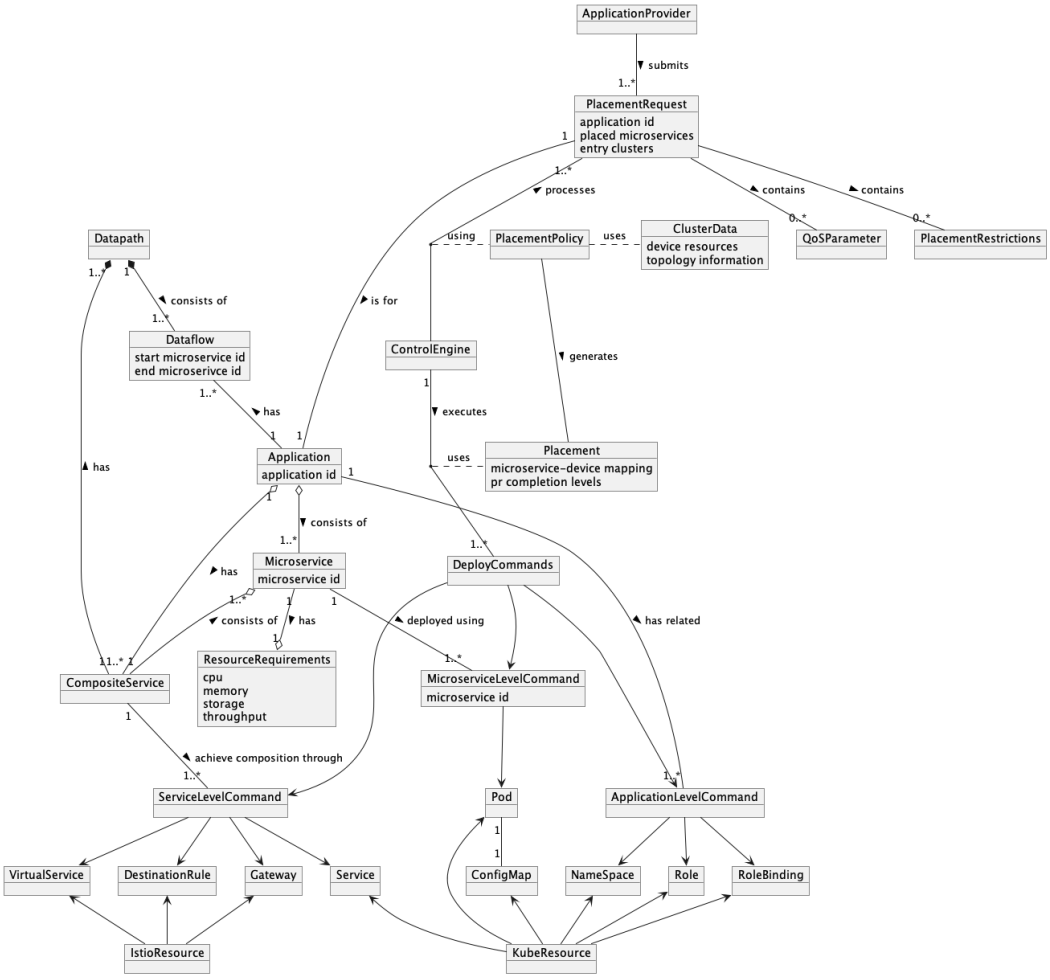

图 4： MicroFog：控制引擎域图

到中央云，或根据资源可用性存储在本地集群中。

5. MicroFog-CE 的应用程序接口

本节重点介绍 MicroFog-CE 提供的三个主要 API，并说明将外部算法集成到 CE 中所需的 API 实现。

• API 1（见图 7）：API 1 用于通过 POST请求接收 PR，请求会被路由至标头中定义的集群。请求以 JSON格式包含与 PR 相关的数据，接收后将使用 Jackson 框架将其映射为基于Java的领域对象。 *"applicationId"*用于标识待部署的应用程序（与 Redis 元数据存储中的可用元数据匹配），而 *"en-tryClusters"*用于指示应用程序所需的流量入口点，这些是请求数据中的必填字段，其他字段则为可选字段。其余字段均为可选字段，可根据相关情况填写。 *"placedMicroservices"* 表示已放置的微服务及其状态。因此，该字段主要用于转发请求；如果应用程序的部分微服务被排除在雾或云中的放置范围之外（即已放置在物联网设备或客户端设备中），也可用于初始 PR 提交。 *"compositionOnlyPlacements* 用于记录需要托管服务级资源的中间集群，以支持跨非相邻集群组合微服务。 *"loadBalancingCompleted"* 的布尔值表示与微服务相关的负载均衡部署信息是否已传输至相关集群，而*" subsetWeights"* 表示用于执行负载均衡策略的设备间相对资源分配。由于微服务之间存在复杂的依赖关系，QoS参数可以在多个粒度级别上定义：针对每个组合服务、微服务之间以及针对每个应用程序 [38]。 *"qosParameters"*字段支持遵循这一要求的详细参数定义。

• API 2（见图 8）：API 2 用于集中式放置模式，通过在标头中定义集群名称来查询每个集群的集群数据，以确保路由。响应返回两种主要类型的数据：1）包含该集群中每个节点资源可用性的数组，其中定义了总资源、查询时的资源使用情况和已分配资源（即以字节为单位的内存和以核心数/虚拟中央处理器表示的 CPU），2）与

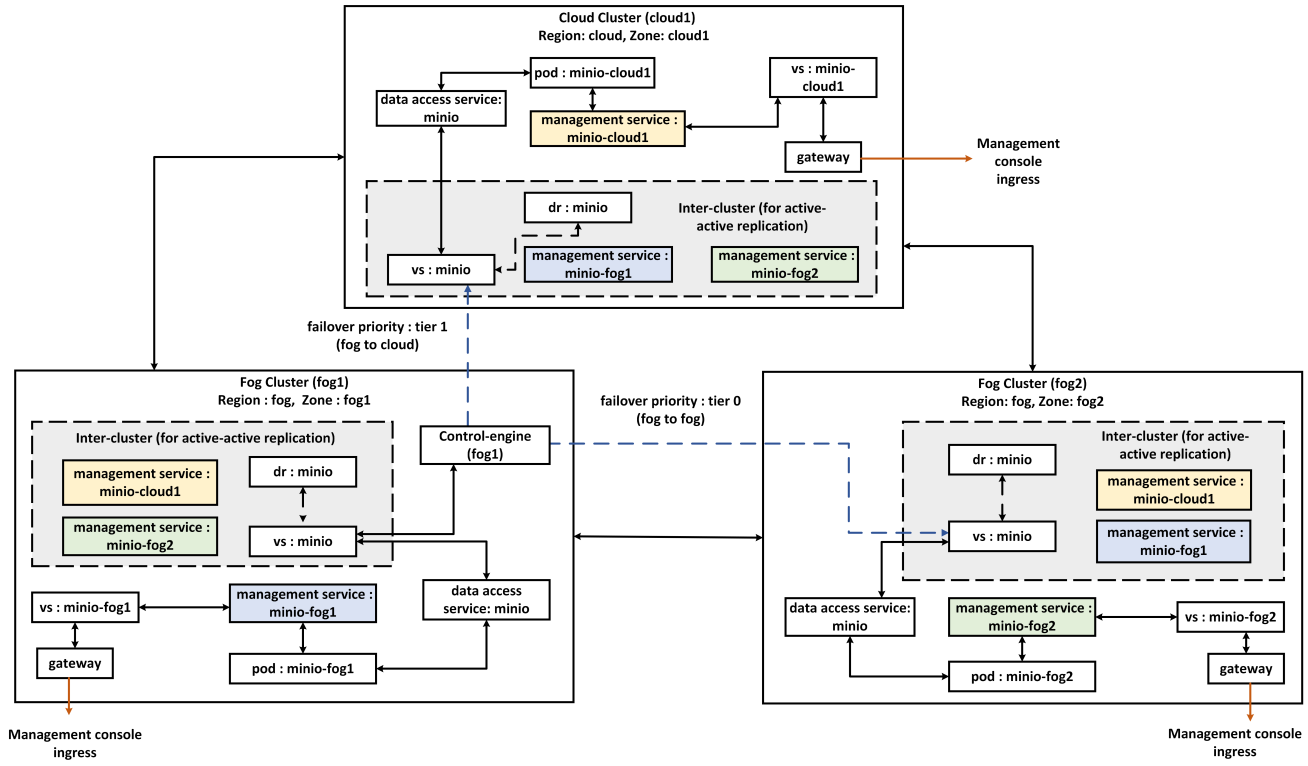

图5： MinIO - YAML 文件存储库 - 部署

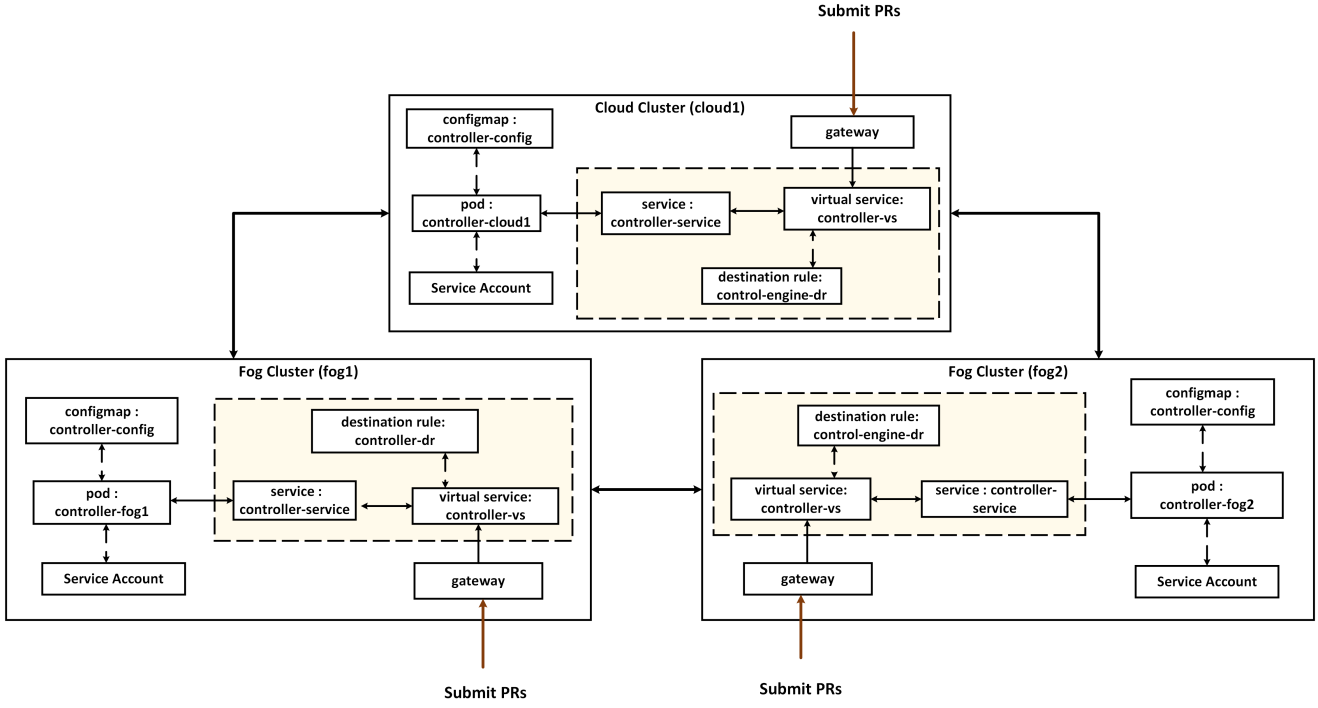

图6： 分布式控制引擎部署

拓扑相关的数据，其中包含相邻雾和云集群的名称。

• API 3（参见图9）：API 3 用于将部署信息传输到标头字段指定的每个集群。对于集中式模式，这包括微服务和雾服务两者服务部署和负载均衡相关的 Istio 资源部署；而在分布式模式下，则仅限于负载均衡相关资源。此 API 还会接受一些额外信息，例如一个布尔标志，用于指明该集群是否为应用程序的入口集群，以便部署 Istio 网关和 VS 资源

以使入口流量能够到达应用程序。请求还包含一个文件，其中列出了需要在集群内部署其服务级资源（即 Kubernetes 服务、Istio 虚拟服务和目的地规则）的微服务（额外的 *MForS* 级别），以确保部署在非相邻集群中的微服务之间保持无缝连接性。

由于使用 Jackson 库在 JSON 数据与 JAVA 领域对象之间进行转换，只需相应地更新相关领域对象，就能轻松修改发送至 API 或从 API 接收的数据。

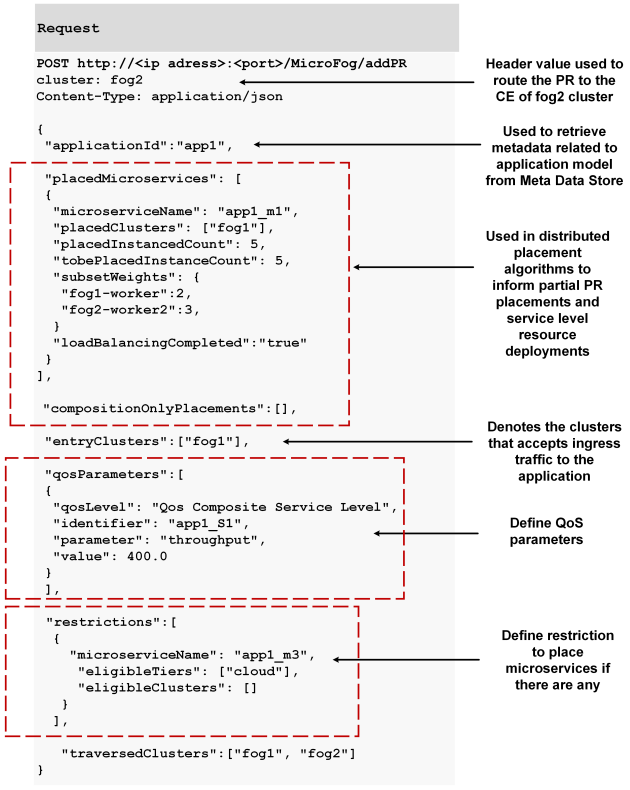

图 7： API 1 - 用于提交拉取请求

6. MicroFog - 评估与验证

本节通过多个用例验证 MicroFog 支持的主要特性和功能。

6.1. 实验设置

6.1.1. 基础设施与 *MicroFog* 设置

为评估 MicroFog 支持的功能，我们构建了一个联邦雾云环境原型，其中包含三个雾计算集群（fog1、fog2 和 fog3）和一个云集群（cloud1）。每个集群都属于独立的网络，并通过负载均衡器相互通信。对于该原型，我们使用 MetalLB [5] 作为负载均衡器，将每个集群暴露给外部。每个集群都是一个

[5]https://metallb.universe.tf/

Request

```
GET http://<ip adress>:<port>/MicroFog/getClusterData
cluster: fog1
```

Response

```
Content-Type: application/json
{
   "clusterName": "fog1",
   "fogDevices": [
      {
          "nodeName": "fog1-worker1",
          "resourcesTotal": {
          "memory": 3878420480,
          "cpu": 2
           },
          "resourcesUsed": {
          "memory": 569229312,
          "cpu": 1.014943195
           },
          "resourcesAllocated": {
          "memory": 2782920704,
          "cpu": 1.500
           }
      },
   ],
   "adjacentFogClusters": [fog2],
   "connectedCloudClusters": [cloud1]
}
```

图 8： API 2 - 用于查询集群信息



图 9： API 3 - 用于提交部署所用的放置结果

独立的 Kubernetes 集群，并通过在各集群中以多主模式实现 Istio 服务网格，维护运行在不同集群中的微服务之间的通信。表 2 总结了每个集群的详情。

表 2联邦雾云基础设施配置

集群 详情	资源	
	CPU（虚拟中央处理器）	内存（GB）
集群 - 雾 1：		
节点1（控制节点）3		6
节点2（工作节点1）	4	9
节点3（工作节点2）	5	16
节点4（工作节点3）	3	8
集群 - 雾 2：		
节点1（控制节点）	3	6
节点2（工作节点1）	3	9
节点3（工作节点2）	2	6
节点4（工作节点3）	4	12
节点5（工作节点4）	4	8
集群 - 雾 3：		
节点1（服务器）	3	6
节点2（代理 0）	2	4
节点3（代理 1）	2	4
集群 - 云 1：		
节点1（控制节点）8		14
节点2（工作节点）	8	14

MicroFog 的主要优势之一是其与云原生技术的兼容性，这使得人们能够快速构建联邦雾云架构的原型，用于开发和评估部署算法，从而克服缺乏公开可用雾资源所带来的限制。为展示这一点，我们使用墨尔本大学 Queensberry Hall 数据中心中可用的虚拟化资源创建 fog1、fog2 和 fog3 集群；该数据中心位于网络边缘。我们还使用通过互联网访问的、位于 ap-southeast-2 的 AWS EC2 实例创建 cloud1。为复现雾集群的行为（其中雾节点通过高带宽局域网链路相互连接），我们将 fog1、fog2 集群实现为 KinD Kubernetes（容器化 k8s）集群，并将 fog3 实现为属于数据中心不同子网的 k3d（容器化 k3s）集群。它们与云集群之间的通信通过广域网进行。

6.1.2. 工作负载创建

由于缺乏多样化的基于微服务的物联网应用基准测试，我们实现了一种工具，用于生成微服务
基于微服务的模拟应用 [6]，以捕捉微服务架构的不同特征，并生成异构应用，用于评估放置策略。该工具提供一个基础微服务作为模板，可通过配置（使用 Kubernetes ConfigMap）创建彼此之间具有多种交互模式（即链式、聚合或微服务候选模式）的微服务，从而构建包含用户可访问的复合服务的基于微服务的应用。此外，还可以配置使用该模板创建的微服务，使其具有不同的处理时间和微服务间消息大小，以模拟异构应用的行为。借助此工具，我们创建了多个包含链式和聚合器交互模式的基于微服务的应用，以评估和验证 MicroFog 框架支持的不同功能。

[6]https://github.com/Cloudslab/MicroFog/tree/main/Workload_生成器

6.1.3. 放置算法

为了突出 MicroFog 支持的主要功能，我们对 [29] 中提出的放置算法进行了调整，并实现了不同的变体。[29] 中的算法旨在将延迟敏感型物联网应用服务尽可能放置在靠近用户的位置，同时满足微服务的资源需求。为此，放置策略从流量入口雾集群开始，逐步移向相邻的雾集群，最后在雾资源不足时考虑云。我们扩展了 [29] 中的策略，纳入吞吐量感知功能：在提交 PR 时可以提供复合服务的吞吐量，放置算法据此计算支持该吞吐量所需的微服务实例数量和资源需求。我们采用 [10] 中提供的计算方法。我们基于此方法创建了三个变体，以评估和验证 MicroFog 的多种配置与功能，具体如下：

1. 版本 1（V1）- 垂直扩展式分布式放置：放置算法从 PR 中检索已放置的微服务，并根据应用程序的 DAG 表示计算下一个要放置的微服务。之后，算法尝试以资源感知的方式在集群内放置该微服务。在此方法中，由于考虑了垂直可扩展性，每个微服务只放置一个实例，以确保其资源分配足以满足吞吐量要求。如果集群没有足够的资源来完成应用放置，则更新 PR 并将其转发到下一个集群，以放置其余微服务。

2. 版本 2（V2）- 水平扩展式分布式放置：此方法与 V1 类似，但支持微服务的水平可扩展性。因此，不再为每个微服务放置单个实例，而是放置多个实例，以满足吞吐量要求。

ces-3. 版本3（V3）- 集中式放置：在此版本中，放置算法维护所有可用集群的视图。收到请求后，算法会选择 PR中定义的一个入口集群。接下来，算法遍历有向无环图，从选定的雾计算集群开始放置微服务；如果没有可用资源，则考虑相邻集群，最后考虑将其放置在云中。

6.2. 用例和结果

6.2.1. 分析*MicroFog*架构的灵活性和可伸缩性

MicroFog架构的灵活性和可伸缩性体现在其能够在分布式多雾、多云环境中运行。我们研究了不同配置下 MicroFog 框架的分布式部署架构，以此加以证明。

- **分布式数据管理与访问：**

本节分析并验证本文提出的用于访问 MinIO YAML 文件存储库和 Redis 元数据存储的部署架构。我们提出的部署架构旨在降低数据存储的延迟并提高其可用性，从而确保应用程序能够可靠地放置和部署。为评估这一点，我们考虑了三种数据访问场景。如图 10(a) 和图 10(b) 所示，我们分别测量了 MinIO YAML 存储和 Redis 元数据存储在各场景下的相对数据检索延迟。我们向部署在 fog1 中的控制引擎提交部署请求，并观察分布式放置模式下的行为。在场景 1 中，两个数据存储均按照图 5 中提出的架构部署在全部 3 个集群中。场景 2 假设 fog1 数据存储不可用，而场景 3 则假设 fog1 和 fog2 中的数据存储均不可用。

结果表明，部署架构能够按预期管理通往数据存储的请求路由。故障转移策略配置为在数据存储发生故障时优先选择距离最近的数据存储。因此，如果所有数据存储均可用，部署在集群 fog1 中的 CE 会访问部署在同一雾计算集群中的数据存储，从而实现最低的数据检索延迟。如果集群内的数据存储不可用，路由策略会优先选择距离最近的相邻雾计算集群，而非云集群；只有在两个雾计算集群中的数据存储均不可用时，才会访问云集群。所获得的延迟值体现了这一行为：雾计算集群之间触发故障转移时，延迟略有增加（场景 2——FO 转至雾）；从雾转移至云时，延迟则相对大幅增加（场景 3——FO 转至云）。因此，所提出的部署架构能够稳健地确保数据访问，同时致力于提升性能。此外，对于资源受限的雾计算集群，将数据存储托管在相邻的资源丰富的雾计算集群或云集群中更为可行，但代价是数据访问性能有所下降。我们提出的架构具有足够的灵活性，能够支持这一行为，并确保联邦式多雾多云环境中的数据访问。

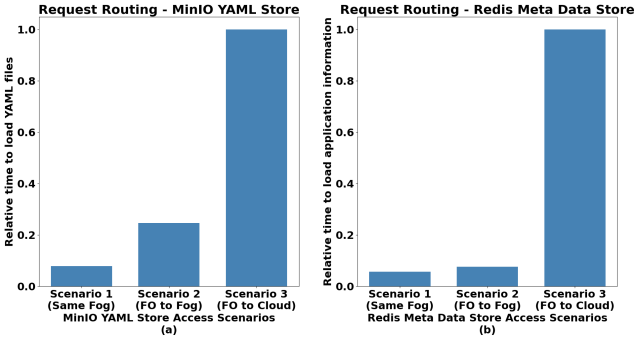

图 10： 数据存储可用性分析

- **对 CE 分布式部署及其运行模式的分析**

MicroFog-CE 旨在实现跨分布式雾和云集群的可扩展部署。为此，CE 支持 CE 的分布式运行模式，在该模式下，所有 CE 独立执行放置算法；也支持集中式模式，在该模式下，主 CE 执行放置算法，并将放置结果发送至各个集群。在这两种方式中，CE 之间的连接性均通过所提出的部署架构（第4.3节）来维持，以实现应用程序的成功放置。

在分布式模式下，PR 可以转发至相邻的雾或云集群，MicroFog-CE 支持集成不同的转发策略，从而为框架用户提供控制分布式放置策略的灵活性。我们通过实现两种转发策略对此进行了演示：1）FP1：如果当前集群没有足够的资源完成 PR 放置，则将 PR 转发至相邻的雾计算集群；2）FP2：如果当前集群没有足够的资源完成 PR 放置，则将 PR 转发至已连接的云集群。为将 PR 路由至选定的集群，会使用目标集群名称更新 PR 转发请求的标头。图 6 中提出的部署架构据此将请求路由至正确的目标。图 11展示了三种场景。在场景 1 中，PR 的入口雾集群包含足够的资源来承载应用程序，因此三种场景中其响应时间最低。场景 2 和场景 3 考虑了入口雾集群没有足够资源承载整个应用程序的情况。场景 2 使用 FP1，将应用程序放置在两个相邻的雾计算集群中；由于雾间通信延迟，其响应时间高于前一种场景。然而，FP1 的表现优于场景 3。场景 3 使用 FP2，请求被转发至云，因此在三种场景中的响应时间最高。上述用例展示了 CE 部署架构应对多个雾和云集群的可伸缩性，以及通过集成转发策略配置分布式放置策略的能力。

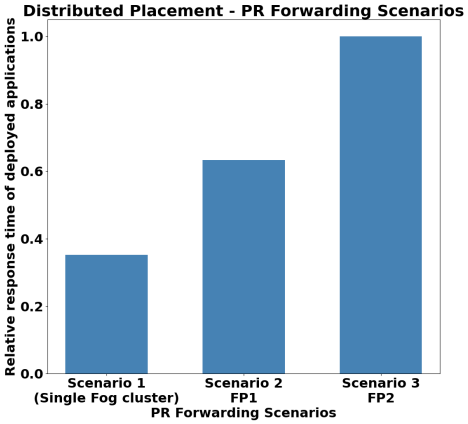

图 11： 分布式放置算法执行MicroFog-CE 也支持集中式放置算法

以及算法执行情况。在图12中，我们考虑了三种放置场景，并分析了 CE 在分布式和集中式运行模式下的应用放置时间。三种场景如下：场景 1——系统同时提交 5 个 PR，其中 3 个以 *fog1* 作为入口集群，另外 2 个以 *fog2* 作为入口集群；场景 2——系统同时提交 10 个 PR，每个集群接收 5 个 PR；场景 3——总计 15 个 PR 同时提交至 fog1、fog2 和 fog3，每个集群接收 5 个 PR。在分布式运行模式下，PR 会提交至其入口集群的 CE；而在集中式模式下，所有 PR 都提交至部署在云中的主 CE。此外，集中式模式使用 V3，而分布式模式使用 V2 作为放置策略。图 12 展示了 PR 部署的总时间，该时间从 CE 接收到 PR 开始计算，直至在事件驱动放置模式下完成应用程序部署。在所有情况下，随着同时处理的 PR 数量增加，分布式模式完成应用放置所需的时间更短。因此，完成时间之间的相对差异会随着 PR 数量增加而扩大。这表明，根据 PR 到达率、放置策略的设计、联邦雾环境的规模等因素，MicroFog-CE 可以配置为使用集中式或分布式放置模式。

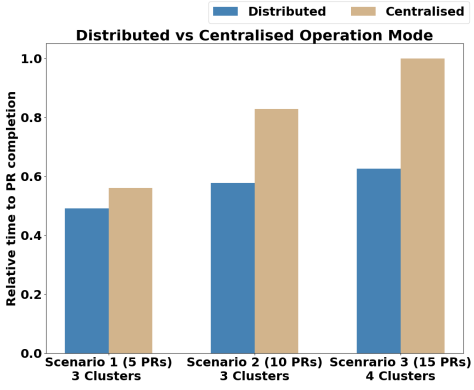

图 12： CE 运行模式分析

- **使用不同 Kubernetes 发行版的分析**

由于资源可用性存在异构性，雾和云集群可以运行不同的 Kubernetes 发行版（即，资源丰富的集群使用 k8s，资源受限的集群使用 k3s）。为了分析 MicroFog 在不同发行版上的运行能力，结果显示，fog3（场景 2）中的 PR 部署时间较短，因为它使用了架构轻量的 k3s；而 fog1（场景 1）的部署时间较长。此外，场景 3 展示了跨集群 PR 放置场景，其耗时比 k3s 集群更长，但由于部署同时跨越两者，耗时比 k8s 部署更短。这表明 MicroFog-CE 能够灵活地在运行不同 Kubernetes 发行版的集群间运行。

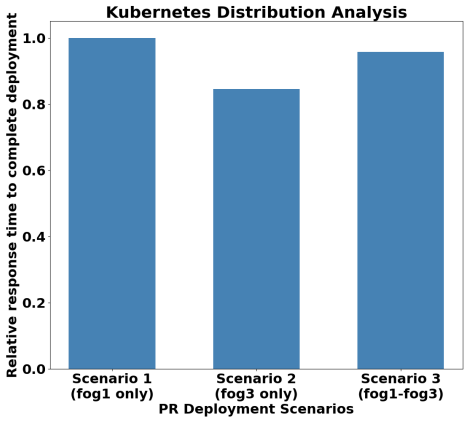

图 13： Kubernetes 发行版分析

上述结果表明，MicroFog 能够处理跨多个集群的放置（可扩展架构），而且 MicroFog-CE 具备可配置性（集成不同的放置算法、转发策略和运行模式），使其能够成功执行放置策略，并在分布式雾和云集群中部署应用。

6.2.2. 联邦雾云部署与微服务组合（服务发现和负载均衡）

微服务架构的主要优势之一，是能够在分布式计算资源之间独立扩展微服务，同时通过服务网格技术确保其动态组合。由于 MicroFog-CE 支持轻松集成多种放置算法，我们实现了 V1 和 V2，以展示可扩展微服务放置的效果，并验证 MicroFog 所支持的动态组合和负载均衡。

我们考虑使用工作负载生成器生成的两个基于微服务的应用的放置：智能健康
第 *2.4.3* 节讨论的护理应用程序（应用ID：hcapp）作为示例物联网应用（见图 2），由两个复合服务组成；另一个是基于有向无环图的应用（应用ID：*app2*），由一个可供用户访问的组合服务组成（见图 14）。该服务包含 4 个微服务，其中 a2m1 和 a2m2 构成链式调用模式，而 a2m2、a2m3 和 a2m4 构成聚合器模式：a2m1 调用 a2m3 和 a2m4，聚合它们的结果并返回给 a2m1 以供进一步处理。表 *3* 记录了放置算法两个版本针对 *app2* 和 hcapp 生成的放置结果。由于 V1 未考虑水平可扩展性，异构雾节点的资源受限特性使放置位置转向云，导致延迟更高，如图 16所示。相比之下，V2 能够利用微服务的水平扩展能力，从而更好地利用有限的雾资源，降低延迟，如图16中可扩展部署所示。结果表明，V2 将

表3示例应用程序（app2 和 hcapp）的生成的部署位置

放置算法	app2		hcapp	
	微服务	已部署节点	微服务	已部署节点
版本1 (V1)	a2m1	fog1-worker2	hcm1	fog2-worker3
	a2m2	fog2-worker4	hcm2	云1-工作节点1
	a2m3	云1-工作节点1	hcm3	cloud1-control-node
	a2m4	云1-工作节点1		
版本2 (V2)	a2m1	fog1-worker2	hcm1	雾节点1-工作节点1, fog1-worker3 分配资源比例 - 1:1
	a2m2	fog1-worker3、fog2-worker1、fog2-worker2 分配资源比例 - 1:2:1	hcm2	雾节点1-工作节点1, fog2-worker1, fog2-worker3 分配资源比例 - 1:2:3
	a2m3	fog2-worker3		
	a2m4	fog2-worker4	hcm1	cloud1-control-node

app2 和*hcapp*占54%。

然而，为确保微服务之间的连接性并维持预期的性能水平，需要在集群之间实现动态服务发现和负载均衡。为此，MicroFog-CE 支持集成新的负载均衡策略。在本实验中，我们实现了一种加权轮询负载均衡策略。MicroFog-CE 的部署规则会根据负载均衡策略的输出部署 Istio 虚拟服务和目标规则。对于上述放置方案，我们通过 Kiali 工作负载图对此进行验证，该图展示了同一微服务的不同水平扩展实例之间的流量分布。表3 显示，对于 *app2* 中水平扩展的微服务 a2m2，其资源分布在分别部署于 fog1-worker3、fog2-worker1 和 fog2-worker2 的实例之间为 1:2:1。得到的图（见图14）显示，a2m2 的流量在两个集群间按 1:3 分配，在 fog2 集群内按 2:1 分配，因此 a2m2 的流量在其三个实例间的分配比例约为 1:2:1。这与加权轮询负载均衡的预期流量分布相符，从而证实 MicroFog 能够自动执行 Istio 资源部署，确保在集群间实现定制的负载均衡能力。图 15进一步展示了这一点，其中反映了 *hcapp*的流量分布。微服务 hcm1（1:1）和 hcm2（1:2:3）的流量分布与它们的资源分布 hcm1（1:1）和 hcm2（1:2:3）相符。

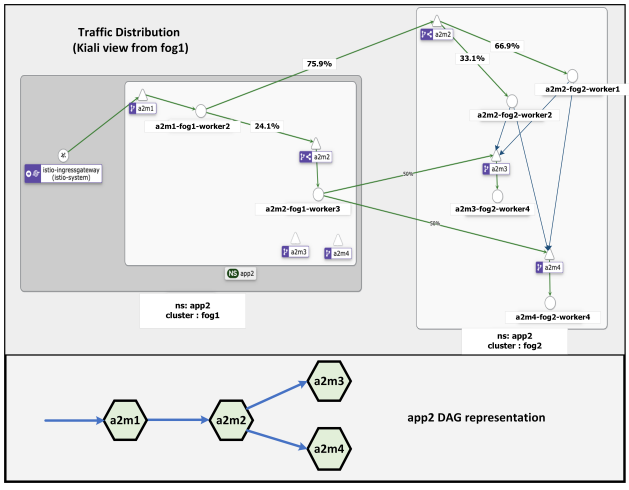

图14： 多集群 Service 服务发现与负载均衡场景 - app2

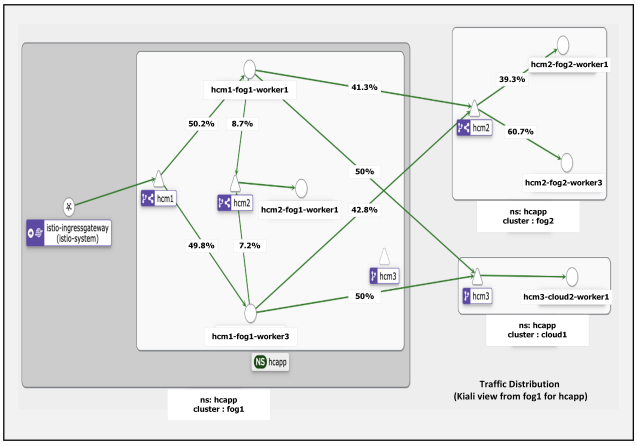

图15： 多集群服务发现与负载均衡场景 - hcapp

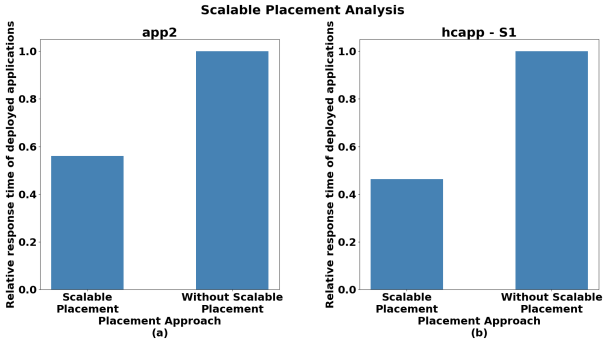

图16： 可扩展的微服务放置

上述用例所获得的结果展示了 MicroFog 支持的不同功能，并验证了 MicroFog 是一种可扩展且易于配置的框架，能够在联邦雾计算环境中部署微服务，并确保跨集群的动态微服务组合。因此，MicroFog 框架可成功用于集成和评估专为放置基于微服务的物联网应用而设计的新型放置算法的性能。

7. 结论与未来工作

在本研究中，我们提出了一个框架，用于在联邦雾环境中可扩展地放置基于微服务的物联网应用程序。该框架具有可扩展性、可扩充性和可配置性，可在启用了 Kubernetes 和 Istio 的多雾多云环境中执行放置算法并部署应用程序。此外，该框架支持集成新颖的放置策略、负载均衡策略和 PR 转发策略。因此，放置算法开发者和物联网应用开发者可以使用该框架在联邦雾环境中部署其应用并监控其性能。此外，该框架支持快速原型开发，且遵循微服务架构开发的应用程序无需进行任何应用级更改，即可使用该框架进行部署。因此，该框架将底层

部署相关功能从用户那里抽象出来，使他们能够更加专注于放置策略开发和物联网应用开发。

由于采用了开源技术、模块化设计和架构，开发者可以轻松扩展该框架，添加新功能。未来，MicroFog 框架还可以进一步改进，例如为跨集群数据传输添加轻量级安全机制，采用可扩展架构来存储和使用可观测性相关数据以改进放置算法，以及支持为应用集成新型容错策略。

软件获取

MicroFog 框架的源代码和文档可通过以下网址获取：https://github.com/Cloudslab/MicroFog

致谢

我们感谢 Melbourne Research Cloud（MRC）提供用于实现 MicroFog 原型的基础设施。

参考文献

[1] M. Fowler 和 J. Lewis。（2014年3月）微服务：这一新架构术语的定义。 [在线]。可访问：https://martinfowler.com/articles/microservices.html[2] R. Mahmud、R. Kotagiri 和 R. Buyya，“雾计算：分类体系、综述与未来方向”，载于《万物互联》。Springer，2018年，第103–130页。[3] M. Goudarzi、M. Palaniswami 和 R. Buyya，“边缘与雾计算环境中的物联网应用调度：分类体系与未来方向”，ACM Computing Surveys，第55卷，第7期，第1–41页，2022年。[4] W. do Espírito Santo、R. d. S. M. Júnior、A. d. R. L. Ribeiro、D. S. Silva 和 R. Santos，“雾计算中基于容器的应用编排的系统性映射”，载于《2019年第15届网络与服务管理国际会议论文集（CNSM）》。IEEE，2019年，第1–7页。[5] P. Farzin、S. Azizi、M. Shojafar、O. Rana 和 M. Singhal，“Flex：用于多雾和多云环境中可扩展服务放置的平台”，载于《2022年澳大拉西亚计算机科学周》，2022年，第106–114页。[6] B. Neha、S. K. Panda、P. K. Sahu、K. S. Sahoo 和 A. H. Gandomi，“关于渗透计算的系统性综述”，ACM Transactions on Internet of Things，第3卷，第2期，第1–30页，2022年。[7] F. Faticanti、F. De Pellegrini、D. Siracusa、D. Santoro 和 S. Cretti，“利用边缘降低吞吐量：雾计算中的应用感知放置”，载于《2019年第6届IEEE网络安全与云计算国际会议（CSCloud）/2019年第5届IEEE边缘计算与可扩展云国际会议（EdgeCom）论文集》。IEEE，2019年，第196–203页。[8] C. Guerrero、I. Lera 和 C. Juiz，“雾计算架构中用于服务放置优化的进化算法的评估与效率比较”，Future Generation Computer Systems，第97卷，第131–144页，2019年。[9] S. Deng、Z. Xiang、J. Taheri、M. A. Khoshkholghi、J. Yin、A. Y. Zomaya 和 S. Dustdar，“资源受限的分布式边缘节点中的最优应用部署”，IEEE Transactions on Mobile Computing，第20卷，第5期，第1907–1923页，2020年。[10] S. Pallewatta、V. Kostakos 和 R. Buyya，“雾计算环境中基于服务质量感知的基于微服务的物联网应用放置”，Future Generation Computer Systems，2022年。

[11] F. Guo、B. Tang 和 M. Tang，“移动边缘计算中微服务组合的时延与成本联合优化”，《万维网》，第 1–29 页，2022 年。[12] R. Mahmud、S. Pallewatta、M. Goudarzi 和 R. Buyya，“Ifogsim2：用于边缘与雾计算环境中的移动性、聚类和微服务管理的扩展版 ifogsim 仿真器”，《系统与软件杂志》，第 111351 页，2022 年。[13] J. Fang 和 A. Ma，“边缘云计算中的物联网应用模块放置与动态任务处理”，《IEEE 物联网期刊》，第 8 卷第 16 期，第 12 771–12 781 页，2020 年。[14] J. Paul Martin、A. Kandasamy 和 K. Chandrasekaran，“Crew：面向雾环境服务放置的成本与可靠性感知鹰鲸优化器”，《软件：实践与经验》，第 50 卷第 12 期，第 2337–2360 页，2020 年。[15] K. Fu、W. Zhang、Q. Chen、D. Zeng、X. Peng、W. Zheng 和 M. Guo，“云边连续体中 QoS 感知且资源高效的微服务部署”，载于《2021年IEEE国际并行与分布式处理研讨会论文集（IPDPS）》。IEEE，2021 年，第 932–941 页。[16] Q. Deng、M. Goudarzi 和 R. Buyya，“Fogbus2：一种轻量级分布式容器化框架，用于将物联网赋能系统与边缘计算和云计算集成”，载于《新兴分布式环境中的大数据国际研讨会论文集》，2021 年，第 1–8 页。[17] D. Santoro、D. Zozin、D. Pizzolli、F. De Pellegrini 和 S. Cretti，“Foggy：雾计算环境中的工作负载编排平台”，载于《2017年IEEE国际云计算技术与科学会议论文集（CloudCom）》。IEEE，2017 年，第 231–234 页。[18] C. T. Joseph 和 K. Chandrasekaran，“跨越鸿沟：微服务软件架构基础与近期进展综述”，《软件：实践与经验》，第 49 卷第 10 期，第 1448–1484 页，2019 年。[19] 访问日期：2 月， 2023 [在线]。 [在线]。可访问：https: //cloud.google.com/distributed-cloud/edge/latest/docs/overview [20] 访问日期：2 月， 2023 [在线]。 [在线]。可访问：https://www.er icsson.com/en/blog/2022/11/service-mobility-in-the-edge-cloud [21] 访问日期：2 月， 2023 [在线]。 [在线]。可访问：https: //www.ibm.com/docs/en/eam/4.2?topic=nodes-edge-clusters [22] O. Skarlat、M. Nardelli、S. Schulte 和 S. Dustdar，“面向 QoS 感知的雾服务放置”，载于《2017年IEEE第一届雾计算与边缘计算国际会议（ICFEC）论文集》。IEEE，2017 年，第 89–96 页。[23] F. Xu、Z. Yin、A. Gu、F. Zhang 和 Y. Li，“多服务雾节点的服务冗余策略与蚁群优化算法”，载于《2020年IEEE第六届计算机与通信国际会议（ICCC）论文集》。IEEE，2020 年，第 1567–1572 页。[24] C. Guerrero、I. Lera 和 C. Juiz，“雾计算中面向性能优化的轻量级去中心化服务部署策略”，《环境智能与人性化计算杂志》，第 10 卷第 6 期，第 2435–2452 页，2019 年。[25] J. L. Herrera、J. Galán-Jiménez、P. Bellavista、L. Foschini、J. Garcia-Alonso、J. M. Murillo 和 J. Berrocal，“面向时间敏感型物联网场景的雾节点、微服务和 SDN 控制器最优部署”，载于《2021年IEEE全球通信大会（GLOBECOM）论文集》。IEEE，2021 年，第 1–6 页。[26] A. Samanta 和 J. Tang，“Dyme：边缘计算赋能物联网中的动态微服务调度”，《IEEE 物联网期刊》，第 7 卷第 7 期，第 6164–6174 页，2020 年。[27] I. Lera、C. Guerrero 和 C. Juiz，“基于图划分的雾计算可用性感知服务部署策略”，《IEEE 物联网期刊》，第 6 卷第 2 期，第 3641–3651 页，2018 年。[28] P. Farhat、H. Sami 和 A. Mourad，“用于按需雾部署时间调度的强化R学习模型”，《超级计算杂志》，第 76 卷，第 388–410 页，2020 年。[29] S. Pallewatta、V. Kostakos 和 R. Buyya，“异构且资源受限的雾计算环境中的基于微服务的物联网应用放置

”，载于《第12届 IEEE/ACM 效用计算与云计算国际会议论文集》，2019 年，第 71–81 页。[30] A. Yousefpour、A. Patil、G. Ishigaki、I. Kim、X. Wang、H. C. Cankaya、Q. Zhang、W. Xie 和 J. P. Jue，“Fogplan：轻量级 QoS 感知动态雾服务配置框架”，《IEEE 物联网期刊》，第 6 卷第 3 期，第 5080–5096 页，2019 年。[31] 访问日期：2 月， 2023 [在线]。 [在线]。可访问：https://fogatlas.fbk.eu/ [32] D. Ermolenko、C. Kilicheva、A. Muthanna 和 A. Khakimov，“基于 Kubernetes 和边缘计算的物联网服务编排框架”，载于《2021年IEEE俄罗斯青年研究人员电气与电子工程会议（ElConRus）论文集》。IEEE，2021 年，第 12–17 页。[33] P. Bellavista 和 A. Zanni，“基于 Docker 容器化在 Raspberry Pi 上部署雾计算的可行性”，载于《第18届分布式计算与网络国际会议论文集》，2017 年，第 1–10 页。[34] S. Tuli、R. Mahmud、S. Tuli 和 R. Buyya，“Fogbus：用于边缘与雾计算的区块链轻量级框架”，《系统与软件杂志》，第 154 卷，第 22–36 页，2019 年。[35] Z. Wang、M. Goudarzi、J. Aryal 和 R. Buyya，“面向实时物联网应用的边缘与雾计算环境中的容器编排”，载于《计算智能与数据分析：IC CIDA 2022 论文集》。Springer，2022 年，第 1–21 页。[36] R. Mahmud 和 A. N. Toosi，“Con-Pi：分布式容器化边缘与雾计算框架”，《IEEE 物联网期刊》，第 9 卷第 6 期，第 4125–4138 页，2021 年。[37] J. Falkner。（2020 年 10 月）IDC 红帽 Quarkus 实验室验证的关键发现。 [在线]。可访问：https://www.redhat.com/en/blog/key-findings-idc-red-hat-quarkus-lab-validation [38] S. Pallewatta、V. Kostakos 和 R. Buyya，“边缘与雾计算中的基于微服务的物联网应用调度：分类体系与未来方向”，arXiv 预印本 arXiv:2207.05399，2022 年。