上海交通大学硕士学位论文

面向无服务器工作流的函数融合与自动配置方法

姓　　名：陈泽彬

学　　号：122033910197

导　　师：尹强

院　　系：计算机系

学 科/专 业：计算机技术专业

申 请 学 位：工程硕士

2025 年 2 月 13 日

A Dissertation Submitted to

Shanghai Jiao Tong University for the Degree of Master

FUNCTION FUSION AND AUTOMATIC CONFIGURATION METHOD FOR SERVERLESS WORKFLOWS

Author: Zebin Chen

Supervisor: Prof. Qiang Yin

Depart of Computer And Science

Shanghai Jiao Tong University

Shanghai, P.R. China

February 13th, 2025

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全知晓本声明的法律后果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

上海交通大学

学位论文使用授权书

本人同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。

本学位论文属于：

□**公开论文**

□**内部论文**，保密□1 年/□2 年/□3 年，过保密期后适用本授权书。

□**秘密论文**，保密____ 年（不超过 10 年），过保密期后适用本授权书。

□**机密论文**，保密____ 年（不超过 20 年），过保密期后适用本授权书。

（请在以上方框内选择打“✓”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

摘 要

无服务器计算作为一种新兴的云计算范式，允许开发者专注于业务逻辑的实现，而无需管理底层服务器和基础设施，因此在云计算领域引起了广泛关注。然而，无服务器工作流在实际应用中仍面临诸多挑战。首先，开发者在开发无服务器应用程序时通常将复杂任务分解为细粒度的、无状态的函数，然而这种细粒度分解与实例之间的一一映射不是最优解，尤其是在嵌套工作流中，可能导致双重计费、级联冷启动等问题，显著影响工作流的性能和成本效益。其次，大部分第三方供应商只为开发者提供捆绑的资源配置选项，这种策略不仅给开发者带来了资源配置负担，而且工作流的资源亲和性差异会显著影响在自动配置过程中所能达到的性能和成本效益。此外，输入敏感型工作流中，不同输入特征（如视频长度、数据规模等）会导致工作流函数对资源需求差异显著，传统资源配置方法无法动态调整，导致资源浪费或性能不足。本文针对无服务器工作流中的部署低效执行问题、资源捆绑分配问题以及输入敏感型工作流的动态资源配置问题展开研究，完成了以下三个工作：

（1）针对无服务器嵌套工作流实际部署的低效执行问题，本文提出了一种动态函数融合方法，该方法的核心思想是通过启发式算法迭代式地融合同步调用的函数，减少同步调用的双重计费时间和远程调用的次数，从而降低成本和端到端时延。在融合阶段结束后还使用适用于工作流的梯度下降算法优化基础设施资源配置，确保在实际负载下实现更佳的性能和成本优化。实验表明，该方法在三种嵌套工作流中能够快速搜索到函数融合最优解，在成本上分别节约 45.5%、51.9% 和 61.0%，进行基础设施优化后比起函数融合最优解在成本上还能进一步优化 33.9%、41.4% 和 49.3%。

（2）针对无服务器工作流中资源捆绑的分配方法所带来的资源浪费和性能不平衡问题，本文提出了一种基于资源解耦的自动配置方法。该方法的核心思想是解耦无服务器函数的内存和 CPU 资源分配策略，根据工作流的资源亲和性，运用优先级调度的思想独立地调整内存和 CPU 资源分配，从而在满足服务水平目标（SLO）的同时，最大限度地降低无服务器工作流的成本。实验表明，该方法在三种不同类型的工作流中保证满足 SLO 要求的同时显著降低了资源浪费，比起基线方法 BO 在成本上平均降低了 44.1%、49.9% 和 34.9%，比起基线方法 MAFF 在成本上平均降低了 31.2%、61.7% 和 45.7%。

（3）针对输入敏感型工作流中不同输入特性对资源需求差异显著的问题，本文提

出了一种基于随机森林模型的动态资源配置方法。该方法的核心思想是运用随机森林模型捕捉输入特征与最佳资源配置之间的复杂关系，建立起资源分配预测模型，依据输入特征动态调整资源分配，以应对不同输入对资源需求的显著差异，最大限度地优化资源利用和成本效率。实验表明，该方法能够根据输入特征自动调整资源配置显著降低成本，比起基线方法 BO 在平均成本上降低 45.6%，比起基线方法 MAFF 在平均成本上降低 55.5%。

关键词：无服务器工作流，函数融合，资源解耦配置，动态资源配置

Abstract

Serverless computing, as an emerging paradigm in cloud computing, allows developers to focus on implementing business logic without the need to manage underlying servers and infrastructure, thus garnering significant attention in the field of cloud computing. However, serverless workflows still face numerous challenges in practical applications. Firstly, when developing serverless applications, developers typically decompose complex tasks into fine-grained, stateless functions. However, this fine-grained decomposition and the one-to-one mapping between instances are not optimal, especially in nested workflows, potentially leading to issues such as double billing, cascading cold starts, and significantly impacting the performance and cost-effectiveness of the workflow. Secondly, most third-party vendors only offer developers bundled resource configuration options. This strategy not only imposes a resource configuration burden on developers but also means that differences in resource affinity across workflows can significantly affect the performance and cost-effectiveness achievable during the auto-configuration process. Additionally, in input-sensitive workflows, varying input characteristics (such as video length, data size, etc.) can lead to significant differences in resource demands for workflow functions. Traditional resource configuration methods cannot dynamically adjust, resulting in either resource wastage or insufficient performance. This paper addresses the issues of inefficient execution deployment in serverless workflows, the problem of bundled resource allocation, and the dynamic resource configuration challenges in input-sensitive workflows. The following three contributions are made:

(1) To tackle the inefficient execution of serverless nested workflows in deployment, this paper proposes a dynamic function fusion method. The core idea is to iteratively fuse synchronously invoked functions using a heuristic algorithm, reducing double billing time and remote invocation frequency, thereby lowering costs and end-to-end latency. After the fusion phase, a gradient descent algorithm tailored for workflows is used to optimize infrastructure resource allocation, ensuring better performance and cost efficiency under actual workloads. Experiments show that this method quickly identifies the optimal function fusion solution for three nested workflows, achieving cost savings of 45.5%, 51.9%, and 61.0%, respectively. After infrastructure optimization, costs are further reduced by 33.9%, 41.4%, and 49.3%

compared to the optimal function fusion solution.

(2) To address resource waste and performance imbalance caused by bundled resource allocation in serverless workflows, this paper proposes a decoupled resource auto-configuration method. The core idea is to decouple memory and CPU resource allocation strategies for serverless functions, independently adjusting memory and CPU allocation based on workflow resource affinity using priority scheduling, thereby minimizing costs while meeting service-level objectives (SLOs). Experiments demonstrate that this method significantly reduces resource waste while ensuring SLO compliance for three types of workflows, achieving average cost reductions of 44.1%, 49.9%, and 34.9% compared to the baseline method BO, and 31.2%, 61.7%, and 45.7% compared to the baseline method MAFF.

(3) To address the significant resource demand variations in input-sensitive workflows due to different input characteristics, this paper proposes a dynamic resource allocation method based on a random forest model. The core idea is to use a random forest model to capture the complex relationship between input features and optimal resource allocation, building a resource allocation prediction model that dynamically adjusts resource allocation based on input features to optimize resource utilization and cost efficiency. Experiments show that this method significantly reduces costs by automatically adjusting resource allocation based on input features, achieving average cost reductions of 45.6% compared to the baseline method BO and 55.5% compared to the baseline method MAFF.

Key words: Serverless Workflows, Function Fusion, Decoupled Resource Configuration, Dynamic Resource Configuration

目 录

插 图

表 格

算 法

第 1 章 绪论

本章首先介绍了时代背景，阐述了无服务器计算的产生背景、核心优势及其应用场景，并分析了无服务器工作流在实际应用中面临的主要挑战。随后，围绕无服务器工作流的低效执行、资源捆绑配置以及输入敏感型工作流的动态资源配置问题，综述了国内外研究现状，并指出现有研究的不足。最后，明确了本文的研究内容与创新点，旨在通过提出动态函数融合、资源解耦配置以及基于输入特征的动态资源配置方法，优化无服务器工作流的性能与成本效益。

1.1 研究背景及其意义

随着互联网的发展，人类社会正在迈进后信息时代，各式各样的网络终端、移动应用在方便人们工作生活需要的同时，产生了海量数据，使得现有的存储和计算设施面临巨大的压力。为了应对这一挑战，谷歌前 CEO Eric Schmidt 在搜索引擎策略会议上的演讲中首次向业界提出了云计算（Cloud Computing）这一概念[1]。现在，世界各国正在加速发展云计算产业，以推动数字经济和提升国家竞争力。美国于 2023 年提出了《国家网络安全战略》[2]，通过基础设施建设、政府采购、构建生态系统以及加强网络安全和云服务安全性等手段，推动云计算技术的进步和产业落地。欧盟于 2023 年提出“数字欧洲计划”[3]，计划通过加大云计算等领域的投入，创新数字解决方案，帮助公民、公共部门和企业共同实现‘欧洲绿色新政’的目标。我国为了实现数字经济的转型和提升国家竞争力，早在“十二五”期间就发布了《国务院关于加快培育和发展战略性新兴产业的决定》[4]，明确了云计算作为战略性新兴产业的重要地位；“十四五期间”发布《“十四五”国家信息化规划》[5]等文件，提出建设泛在智联的数字基础设施、推动自主创新、培育市场、加强国际合作以及促进产业数字化转型等措施，推动云计算产业的快速发展。

云计算作为一种植根于搜索引擎平台设计的计算技术，其计算服务由通过网络连接的大量低成本计算单元提供[6]。从服务类型的视角看，云计算模式可以分为基础设施即服务（Infrastructure as a Service, IaaS）、平台即服务（Platform as a Service, PaaS）和软件即服务（Software as a Service, SaaS）[7-8]。IaaS 提供了虚拟机、存储和网络等基础设施资源，用户需要自行配置和管理这些资源。PaaS 进一步简化了开发环境，提供了应用部署和管理平台，但用户仍然需要关注应用的扩展和维护。SaaS 直接为终

端用户提供按需付费的软件服务，用户不再需要关注基础设施和平台，但也局限于供应商所开发的软件功能。

除开 SaaS 这一通过互联网提供软件的模式，传统的云计算模式通常选择 IaaS 和 PaaS，这种选择的缺点是开发者必须自己管理虚拟机，基本上要么成为系统管理员，要么与云供应商合作设置环境[9]。低级虚拟机的繁琐管理激发了客户想要拥有更简单的云应用的需求，他们希望为新应用找到一条更简单的通往云的路径，对这些需求的认识促使亚马逊在 2014 年推出了 AWS Lambda 服务[10]。AWS Lambda 作为最早的无服务器计算（Serverless Computing）平台，提供了云函数这一无服务器计算的核心功能——用户只需编写代码并将所有服务器配置和管理任务交给云供应商。AWS Lambda 的发布引起了对无服务器计算的广泛关注。

无服务器计算作为一种新兴的云计算范式，并不是指没有服务器参与计算，而是指云供应商负责管理服务器资源，包括配置、扩展和维护等繁琐的工作。开发者只需编写和上传代码，云平台会自动处理资源的分配和释放，从而实现按需付费的模式。云供应商提供的云函数，被称为函数即服务（Function as a Service, FaaS）产品，代表了无服务器计算的核心。除此之外，云供应商还提供了诸如消息队列、非关系型数据库等后端即服务（Backend as a Service, BaaS）产品，以满足特定的应用需求[11]。简而言之，目前最被广泛接受的观点为：无服务器计算 = FaaS + BaaS[12]。

无服务器计算其核心优势在于通过抽象化底层基础设施，使开发者无需关注服务器的管理与维护，从而能够专注于业务逻辑的实现。其按需付费模式显著降低了资源浪费，提高了成本效益，尤其适用于需求波动较大的应用场景。此外，无服务器架构具备高可扩展性和弹性，能够自动应对流量峰值，确保系统稳定性[13]。基于以上各种优势，无服务器计算在多个领域展现了广泛的应用潜力[14]，包括微服务架构[15-16]、事件驱动系统[17]、人工智能与机器学习[18]以及物联网（IoT）设备管理[19]等领域。

随着无服务器计算的兴起，各大云服务提供商纷纷推出了自己的无服务器产品，这些平台不仅提供了丰富的功能和工具，还通过不断的技术创新和优化，推动了无服务器计算的发展。AWS Lambda 作为最早的无服务器计算平台，提供了强大的功能和灵活的配置选项，支持多种编程语言，如 Java、Python、Go 和 Node.js 等流行的编程语言。除了 AWS Lambda 这家典型无服务器平台，各厂商还陆续推出了诸如 Google Cloud Functions[20]、IBM Cloud Functions[21]、阿里云函数计算[22]等无服务器计算服务产品。

1.2 研究问题与挑战

无服务器计算最初主要应用于单函数应用，开发者通过调用单个无状态的函数来实现特定的业务逻辑。这种模式的优势在于其简单性和快速部署能力，然而随着应用复杂性的增加，单函数应用逐渐暴露出其局限性，尤其是在处理复杂业务流程和大规模数据处理任务时。为了应对这些挑战，单函数应用逐渐演变为工作流应用，其中多个函数通过事件驱动的方式协同工作，形成复杂的业务流程[23-24]。工作流应用的兴起标志着无服务器计算从简单的函数调用扩展到更复杂的任务编排。例如，数据处理流水线、实时分析和大规模计算任务等场景，都需要多个函数之间的紧密协作和高效调度。这种演进不仅提高了系统的灵活性和可扩展性，还为开发者提供了更强大的工具来应对多样化的应用场景。通过将复杂的业务逻辑分解为多个独立的无服务器函数，开发者能够实现更细粒度的控制和更高效的资源利用。当前的主流平台如 AWS Step Functions[25]、Azure Durable Functions[26]和 Google Cloud Workflows[27]提供了对工作流的支持，允许开发者通过显式的工作流规范来编排多个函数的执行顺序和依赖关系。

本文围绕无服务器工作流，首先研究无服务器工作流尤其是嵌套工作流的低效执行问题，旨在为嵌套工作流解决双重计费、级联冷启动等问题，提高工作流的性能和成本效益。其次，由于当前云供应商通常将无服务器函数资源配置问题交由开发者管理，且捆绑资源分配策略，本文研究了无服务器工作流的资源解耦配置问题，以解决资源捆绑分配带来的资源浪费和性能不平衡问题。此外，针对输入敏感型工作流中不同输入特性对资源需求差异显著的问题，本文进一步研究了基于输入特征的动态资源配置方法，以优化资源利用和成本效率。

当下，这些研究问题面临着许多挑战。

（1）无服务器工作流的低效执行挑战。无服务器工作流的低效执行主要是由于冷启动、远程调用开销、双重计费等问题导致的。在无服务器计算环境中，冷启动是一个显著的性能瓶颈，其含义是指函数在一段时间未被调用后，需要重新初始化的过程，包括设置执行环境、加载函数代码和初始化依赖项，从而引入明显的延迟。在涉及多个函数的工作流中，这种延迟会累积，显著增加整体响应时间。无服务器函数之间的通信通常通过网络进行，这种通信方式引入了显著的远程调用开销，尤其对于短运行时间的函数，网络延迟可能超过函数执行时间本身。在复杂的工作流中，这种开销会迅速累积，影响整体性能并增加运营成本。双重计费是无服务器工作流，尤其是嵌套工作流中的一个重要的挑战，指当一个函数同步调用另一个函数并等待其响应

时，两个函数的执行时间都会被计费，即使调用函数处于等待状态。这种计费方式在复杂的工作流中尤为突出，导致成本显著增加。

（2）无服务器工作流的资源捆绑分配挑战。无服务器工作流的资源捆绑配置挑战来自于现有平台资源配置机制的不灵活性以及细粒度函数对资源需求的多样性。当前的无服务器平台大多采用内存与 CPU、网络带宽等资源紧密耦合的配置机制，这种配置方式虽然简化了资源分配过程，但也限制了资源的灵活性，并不适用于所有类型的无服务器工作流负载。例如，内存密集型工作流和计算密集型工作流对资源的需求存在显著差异，单一的配置机制无法满足工作流多样化的需求，会带来潜在的资源浪费。此外，无服务器工作流包含若干不同业务逻辑的细粒度函数，不同函数所需的资源配置可能存在巨大差异，大多数开发者难以根据工作负载准确评估细粒度函数的资源需求，尤其是对于无服务器背景下经常突变的工作负载。为了解决细粒度函数的资源配置问题，开发者通常只能过度配置以求得应用程序的稳定性。而且资源分配决策可以显著影响无服务器工作流的性能和执行成本，选择错误的配置可能导致执行时间和执行成本比最佳配置差若干倍的情况。对于无服务器工作流来说，无论是过度配置还是错误配置，都会造成巨大的执行成本浪费。

（3）输入敏感型工作流的动态资源配置挑战。输入敏感型工作流中，不同输入特性（如视频长度、数据规模等）对资源需求差异显著，传统资源配置方法无法动态调整，导致资源浪费或性能不足。例如，在视频分析工作流中，长视频输入通常需要更多的计算资源来处理，而短视频输入所需的资源较少。然而，现有的资源配置方法通常采用固定的资源分配策略，无法根据输入特性动态调整资源配置，导致在处理长视频输入时资源不足，而在处理短视频输入时资源浪费。这种静态资源配置方式不仅降低了资源利用效率，还增加了运行成本。因此，如何根据输入特性动态调整资源配置，以优化资源利用和成本效率，是输入敏感型工作流面临的主要挑战。

1.3　国内外研究现状

无服务器工作流作为一种新兴的计算范式，近年来在学术界和工业界引起了广泛关注。然而，其在实际应用中仍面临诸多挑战，尤其是在低效执行、资源捆绑配置以及动态资源配置等方面。针对这些问题，国内外研究者开展了大量研究，并取得了一系列重要进展。本节将从无服务器工作流的低效执行优化、资源捆绑配置以及动态资源配置三个方面，系统梳理现有研究成果，分析其优势与不足，为本文的研究提供理论基础和技术参考。

1.3.1 无服务器工作流低效执行优化现状

为了解决无服务器工作流实际部署的低效执行问题，现有研究在无服务器计算的冷启动管理、优化远程调用开销和函数融合方面取得了显著进展。

冷启动管理 无服务器计算中的冷启动管理一直是研究的热点，为缓解冷启动延迟的副作用，研究者们提出了多种解决方案，着眼于打破单个无服务器函数的冷启动管道中的潜在瓶颈。Mohan[28]等人通过使用暂停容器池移除了网络命名空间构造管道中的瓶颈，即预先构建这些资源，然后动态地将它们重新关联到基准容器中，显著减少了冷启动时间。Oakes[29]等人分析了通过 Python 包索引分发的库的包类型和可压缩性，提出了一种基于 OpenLambda 的包感知计算平台去共享用户级库，通过优化库的加载和初始化过程进一步缩短了冷启动时间。缓慢的应用程序和容器初始化会损害无服务器平台上的常见情况延迟，Oakes[30]等人提出了 SOCK，通过分析 Linux 容器原语识别与存储和网络隔离相关的可扩展性瓶颈，制定精简容器以加快启动速度，显著提升了冷启动性能。

在解决了单个函数的冷启动问题后，研究者们开始关注函数链中的级联冷启动问题。级联冷启动是指在一个函数链中，前一个函数的冷启动延迟会传递给后续函数，导致整体延迟显著增加。Akkus[31]等人提出了 SAND，这项工作采用多级故障隔离，通过共享隔离沙箱来减少冷启动延迟。Bermbach[32]等人提出了一种在保持函数为黑盒的同时缓解级联冷启动的方法，通过使用外部提示管理器预启动工作线程来防止级联冷启动。Daw[33]等人提出了 Xanadu，采用及时资源配置技术，防止级联冷启动，同时保持高资源使用效率，其核心思想在于通过采用函数链执行路径的概率模型，在链的执行推进前推测性地部署资源。

尽管冷启动管理的研究取得了显著进展，但现有方法仍存在许多不足。最常见的预构建资源池和库共享技术虽然有效，但增加了系统的复杂性和维护成本。这些方法需要在系统中引入额外的组件（如容器池、库管理系统），导致部署和运维的复杂性增加，还可能导致与现有平台的兼容性问题，增加了迁移和部署的难度。一些冷启动缓解策略虽然在一定程度上减少了延迟传递，但在高并发和复杂依赖场景下效果有限。例如通过预启动容器实例的思路，其对函数依赖关系的准确预测依赖于复杂的概率模型，实际应用中可能难以保证预测的准确性。

优化远程调度开销 在优化远程调度开销方面，现有工作主要集中在减少函数间的数据传递延迟和优化函数的放置策略。Mahgoub[34]等人提出了 SONIC，旨在通过联合优化数据共享方式（例如在 VM 内共享文件或在 VM 之间复制文件）和工作流

中函数的放置来减少函数交互延迟。Barcelona-Pons 等人[35]提出了 Crucial，通过在修改后的 Infinispan 内存数据网格上引入分布式共享对象（DSOs）来应对在无服务器环境中构建有状态应用的挑战，改进了高度可并行化函数的状态共享。Klimovic[36]等人提出了 Pocket，通过提供弹性分布式数据存储来解决无服务器分析中的中间数据交换问题，通过动态调整资源规模并运用多种存储技术，在保证性能的同时降低成本。

上述所有系统都支持分布式 FaaS 应用程序，其中交互函数在不同机器上执行，而有些工作旨在通过在单台机器上执行工作流来减少函数交互延迟。Shillaker 等人[37]提出了 Faasm，通过基于 WebAssembly 的软件故障隔离减少无服务器计算的资源开销，允许在同一机器上部署的函数直接共享内存，且通过从快照恢复函数来最小化初始化时间。Jia[38]等人提出了 Nightcore，通过在同一物理机器上共存并使用共享内存来实现高效数据共享，从而加速无状态微服务。

在优化远程调用开销方面，现有工作虽然通过共享内存、分布式数据存储等技术减少了延迟，但仍存在显著的改进空间。基于文件共享的系统在数据传递效率上存在瓶颈，尤其是在大规模数据交换场景下，文件系统的性能瓶颈可能导致延迟增加。部分系统如 Crucial 和 Nightcore 要求应用程序进行修改或编译为特定格式，限制了其普适性。此外，远程数据存储的访问开销仍然是性能优化的主要障碍，尤其是在跨平台和多云计算场景中，例如 Pocket 对远程数据存储的依赖可能导致网络延迟较高的场景下的性能波动。

函数融合 冷启动优化和通信开销优化取得了显著进展，但从根本上讲，大多数方法的核心在于通过某种共享机制或预测机制去减少这些不必要的开销。这些方法往往引入了复杂的额外工作，增加了系统的复杂性，而且无法解决无服务器工作流中的嵌套工作流的双重计费问题。因此，研究者们将目光转向函数融合，期望通过这一更简单、更普适的方法来进一步优化无服务器系统的性能和成本。

Elgamal[39]等人提出了 Costless，一种函数融合和放置框架，特别适用于物联网-云场景，其中图像处理管道在边缘（AWS Greengrass）和云（AWS Lambda）上调度。该算法使用约束最短路径算法，在满足延迟阈值的前提下，最小化成本。Schirmer[40]等人提出了 Fusionize，一种反馈驱动的函数融合技术，旨在减少 FaaS 提供商的计费并提高端到端应用程序延迟。该方法融合串联函数以避免双重计费，同时保持异步调用函数为单独的融合组。Costless 和 Fusionize 都假设单一输入大小和单一资源配置。Mahgoub[41]等人提出了 WiseFuse，通过使用性能模型预测函数的调用性能，结合函数融合、捆绑、预热和资源分配等技术最小化延迟和成本。Khochare[42]等人提出了

XFaaS，一种跨平台编排无服务器工作流的技术，适用于涉及 AWS 和 Azure 的多云计算提供商场景。其函数放置和融合技术能够将子 DAG 放置在单个云服务提供商上，或根据用户指定的数据监管约束和最小化端到端延迟和成本的目标，将子 DAG 分区到不同的云服务提供商。Sheshadri[43]等人提出了 RightFusion，一种新颖的函数融合技术，根据应用程序输入和 QoS 要求在边缘-云资源池上执行函数融合。该平台能够适当调整资源实例大小，并在满足 QoS 要求的同时执行成本高效的函数融合，还利用异构资源成本差异来选择融合函数的部署资源。

现有函数融合的研究方法在适应线上工作负载、基础设施优化和跨平台场景方面存在局限性。Costless、Fusionize 等技术在单一输入大小下表现优异，但在处理多变的线上数据输入时可能无法有效适应。XFaaS 和 RightFusion 等跨平台融合技术虽然在多云计算场景中表现优异，但其对固定资源大小的依赖限制了其灵活性，可能无法进一步挖掘基础设置配置优化带来的性能和成本的进一步提升。WiseFuse 虽然通过性能模型和多种部署策略去尽可能最大化函数融合带来的性能提升与成本优化，但其策略基于 AWS 平台构建，跨平台迁移该工作在带来大量工作量的同时无法保证其性能。

1.3.2 无服务器工作流资源捆绑配置研究现状

商业云提供商已经开发了一些系统，为用户提供运行其任务的合适配置参数建议，例如 AWS Lambda Power Tuning[44]通过测试不同内存配置下的函数执行性能和成本，自动选择最优的资源配置。但这些云供应商提供的工具或系统内置的配置调度，都有其各自的缺点，且难以应对复杂多变的无服务器工作流。为了解决无服务器工作流资源配置问题，研究者们提出了多种方法，主要分为两大类：一类是通过优化单个函数的资源配置来为工作流找到最优资源配置，另一类是考虑函数间的耦合关系来为工作流找到最优资源配置。

函数资源配置 调整无服务器函数的资源配置的方法主要分为两大类：依赖先验分析的离线方法和依赖运行时监控的在线方法。

依赖先验分析的系统通常在不同资源配置下执行单个函数，并使用典型输入数据来了解其性能。生成的性能配置文件用于在生产中配置函数的资源，以满足响应时间 SLO 或成本要求。这些方法可以根据用于确定资源配置的算法进一步分类，如使用二进制搜索、梯度下降搜索、贝叶斯优化等。Zubko 等人[45]提出了 MAFF 框架，采用线性、二分和梯度下降算法等多种优化算法来自动找到无服务器函数的最佳内存配置。Akhtar[46]等人提出的 COSE 是一个基于贝叶斯优化的框架，利用统计学习技

术，通过收集样本并预测未见配置下的成本和执行时间，从而构建无服务器函数的性能模型。该框架由性能建模器和配置查找器两个主要组件组成，性能建模器负责学习成本、运行时间与配置参数之间的关系，而配置查找器则根据学习到的模型，选择满足延迟约束的最小成本配置。受制于云供应商提供的平台，这些工作为函数寻找的配置均为内存配置。Bilal[47]等人提出了一种原则性的方法，通过解耦内存和 CPU 分配以及启用不同虚拟机类型，来优化性能和成本。

离线函数配置方法依赖于“典型工作负载”的定义，这对于输入高度可变的函数可能难以实现。由于无服务器函数的输入通常具有动态性和多样性，离线方法难以准确预测所有可能的输入场景，导致其在实际应用中的适应性受限。其次，离线方法通常需要对函数进行多次性能测试，这不仅增加了开发和部署的复杂性，还可能导致额外的成本开销。此外，离线方法在处理复杂工作流时可能面临挑战，因为工作流中的函数间耦合关系和依赖性难以通过单个函数的离线测试来准确建模，导致整体工作流的资源配置优化效果不佳。

在线方式构建性能模型的系统依赖历史数据或实时监控数据。一些方法被动监控执行，而其他方法则为函数运行分配不同配置，直到性能模型完成。Cordingly[48]等人设计了 SAAF，收集在线性能数据并结合 Linux CPU 时间核算原则和多元回归分析预测函数运行时间，构建性能预测模型来估计 FaaS 工作负载在异构环境中的运行时间、运行成本，进而优化无服务器函数配置。Eismann[49]等人提出了 Sizeless，使用单一资源大小的监控数据来预测无服务器函数的最佳资源大小，该方法不需要专门的性能测试，而是通过生成合成无服务器函数并监控其在不同内存大小下的执行时间和资源消耗，构建一个多目标回归模型。Yu[50]等人介绍了 FaaSDeliver，通过实时监控每次函数调用的性能和资源消耗来收集数据，再基于树结构 Parzen 估计器构建性能模型，以区分最优或接近最优的函数交付策略，为每次函数调用生成预计成本较低的函数交付策略。

在线函数配置方法需要在应用程序的整个生命周期内分配额外资源来收集和处理监控信息，这通常会导致无服务器应用程序的总成本配置文件次优，例如 FaaSDeliver 和 Sizeless 等方法虽然通过实时监控数据优化了资源配置，但其对额外资源的依赖增加了运行成本。此外，在线方法在处理复杂工作流时可能面临性能瓶颈，因为工作流中的函数间耦合关系和依赖性增加了监控和优化的复杂性，导致实时调整的难度加大。最后，在线方法的动态调整机制可能导致资源配置的频繁变化，增加了高并发和大规模工作流场景下系统的复杂性和不稳定性。

工作流资源配置 许多研究者发现，想要为工作流资源配置取得更优的效果不仅需要考虑单个函数的内存和成本，还需要考虑函数间的各类耦合关系。Safaryan[51]等人提出的 SLAM 通过分布式追踪技术检测 FaaS 函数之间的关系，并使用最大堆方法建模每个函数在不同内存配置下的执行时间，从而估计应用程序的总体执行时间。Raza[52]等人改进了 COSE，使其适用于无服务器工作流，全局优化整个工作流的性能和成本。Wen[53]等人介绍了 StepConf，使用分段拟合模型来预测不同配置下函数的计算延迟，将工作流配置优化问题形式化为每个函数步骤的决策问题，使用基于关键路径的启发式优化算法来动态优化配置，以满足实时需求和 SLO。Feng[54]等人提出了 PLOEA，使用一个集成的多任务专家分类器，分析个体和共同的资源使用模式，考虑分配紧急性、函数间通信亲和性以及服务器的多核架构以准确估算异构工作流应用程序的资源配置。

现有工作流资源配置方法在处理复杂工作流时可能面临性能瓶颈，因为工作流中的函数间耦合关系和依赖性增加了资源配置的复杂性。例如 SLAM 和 StepConf 等方法虽然通过局部最优或启发式算法优化了资源配置，但其对复杂工作流的适应性仍有限，尤其是在处理包含条件分支和动态依赖的工作流时。而且现有方法在处理异构工作流时可能存在局限性，因为不同函数对资源的需求差异较大，导致统一的资源配置优化策略难以满足所有函数的需求。此外，现有方法在处理动态工作负载时可能存在滞后性，因为工作负载的变化速度可能超过资源配置的调整速度，导致优化效果无法及时反映实际需求。

1.3.3 无服务器工作流动态资源配置研究现状

无服务器工作流通过将复杂任务分解为多个函数调用阶段，提供了高度的模块化和细粒度的扩展能力。然而，由于工作流中各个阶段的资源需求差异较大，且输入特征对资源消耗的影响显著，传统的静态资源配置方法难以满足性能、成本和资源利用率的最优平衡。因此，研究人员将目光放到了无服务器函数的动态资源配置上，旨在根据实时输入特征和工作流需求，动态调整资源分配，以提升性能并降低成本。

动态资源资源配置方法 现有研究主要通过构建输入特征与资源配置之间的关系模型，实现动态资源配置。Zhou[55]等人介绍了 Aquatope，一个针对多阶段无服务器应用的 QoS 和不确定性感知资源调度器，通过使用可扩展的贝叶斯模型预测函数调用并预先预热容器，同时根据函数粒度分配资源，以满足复杂工作流的端到端 QoS 要求，并最小化资源成本。Moghimi[56]等人介绍了 Parrotfish，一个基于参数回归的无服务器函数资源配置优化工具，利用参数回归技术构建函数执行时间与内存大小之

间的内在关系模型，准确预测不同资源配置下的执行时间，并动态更新其参数回归模型，实现快速推荐最优的资源配置。Bhasi[57]等人提出了 Cypress，一个输入大小敏感的无服务器资源管理框架，通过输入大小敏感的请求批处理和请求重排序等策略，优化容器资源分配，减少资源消耗并确保高 SLO 合规性。Sinha[58]等人提出了 Shabari 框架，通过延迟资源分配决策至函数输入可用时，使用在线学习代理根据函数输入的特征来正确调整每次函数调用的大小，并做出冷启动感知的调度决策，从而减少服务级别目标（SLO）违规并提高资源利用率。

尽管现有研究在无服务器工作流动态资源配置方面取得了显著进展，但仍存在一些不足。许多方法虽然考虑了输入特征，但对输入属性的理解仍较为局限，例如仅依赖输入大小而忽略其他关键属性（如数据复杂度或类型），导致资源配置不够精确。其次，Parrotfish 虽然通过参数回归降低了探索成本，但在处理多阶段工作流时，缺乏对跨函数依赖性和全局资源优化的考虑，可能导致局部最优而非全局最优的资源配置。此外，现有方法在冷启动优化方面仍有改进空间，Shabari 引入了冷启动感知调度，但其对冷启动的预测和缓解机制仍依赖于启发式方法，未能完全消除冷启动对性能的影响。

1.4　研究内容及创新点

本文针对无服务器工作流中的部署低效执行问题、资源捆绑分配问题以及输入敏感型工作流的动态资源配置问题展开研究，旨在提升工作流的性能和成本效益。首先，开发者在开发无服务器应用程序时通常将复杂任务分解为细粒度的、无状态的函数，然而这种细粒度分解与实例之间的一一映射不是最优解，尤其是在嵌套工作流中，可能导致双重计费、级联冷启动等问题，显著影响工作流的性能和成本效益。其次，大部分第三方供应商只为开发者提供捆绑的资源配置选项，这种策略不仅给开发者带来了资源配置负担，而且工作流的资源亲和性差异会显著影响在自动配置过程中所能达到的性能和成本效益。此外，输入敏感型工作流中，不同输入特征（如视频长度、数据规模等）会导致工作流函数对资源需求差异显著，传统资源配置方法无法动态调整，导致资源浪费或性能不足。

本文针对这些挑战展开研究，主要贡献如下：

（1）针对无服务器嵌套工作流中的低效执行问题，提出了一种动态函数融合方法。该方法通过启发式算法融合同步调用的函数，减少同步调用的双重计费时间和远程调用的次数，从而显著降低成本和端到端时延。在融合阶段结束后还使用适用于

工作流的梯度下降算法优化基础设施配置，确保在实际负载下实现最佳的性能和成本优化。根据此方法设计了一个反馈驱动的函数融合框架，能够根据在线数据动态调整工作流部署实例。最后通过实验验证了所提出的函数融合框架在三个典型嵌套工作流中的有效性，尤其是在减少双重计费、级联冷启动和远程调用开销方面表现突出。实验证明该框架能够快速搜索到函数融合最优解，在成本上分别节约 45.5%、51.9% 和 61.0%，进行基础设施优化后比起函数融合最优解在成本上还能进一步优化 33.9%、41.4% 和 49.3%。

（2）针对无服务器工作流中资源捆绑的分配方法所带来的资源浪费和性能不平衡问题，提出了一种基于资源解耦的自动配置方法。该方法打破了传统无服务器平台中内存与 CPU 资源的紧耦合配置模式，通过独立调整内存和 CPU 资源，能够根据工作流的资源亲和性进行精细化配置，显著降低了资源浪费和运行成本。最后通过实验验证了该方法在三种不同类型的工作流中保证满足 SLO 要求的同时显著降低了资源浪费，比起基线方法 BO 在成本上平均降低了 44.1%、49.9% 和 34.9%，比起基线方法 MAFF 在成本上平均降低了 31.2%、61.7% 和 45.7%。

（3）针对输入敏感型工作流中不同输入特性对资源需求差异显著的问题，本文提出了一种基于随机森林模型的动态资源配置方法。该方法的核心思想是运用随机森林模型捕捉输入特征与最佳资源配置之间的复杂关系，建立起资源分配预测模型，依据输入特征动态调整资源分配，以应对不同输入对资源需求的显著差异，最大限度地优化资源利用和成本效率。最后通过实验验证了该方法能够根据输入特征自动调整资源配置显著降低成本，针对典型的输入敏感型工作流，该方法能够根据输入特征自动调整资源配置显著降低成本，比起基线方法 BO 在平均成本上降低 45.6%，比起基线方法 MAFF 在平均成本上降低 55.5%。

1.5 论文组织架构

本文共分为五个章节，具体内容安排如图1.1所示：

第一章绪论。本章首先介绍了无服务器计算的产生背景、核心优势及其应用场景，阐述了无服务器计算在云计算领域的重要地位及其广泛的应用前景。接着，分析了无服务器工作流在实际应用中面临的主要挑战，特别是低效执行问题、资源捆绑配置问题以及输入敏感型工作流的动态资源配置问题。随后，梳理了国内外在无服务器工作流优化领域的研究进展，重点回顾了低效执行优化、资源捆绑配置和动态资源配置等方面的研究成果，并指出现有研究的不足。最后，总结了本文的研究内容与创

图 1.1　论文架构
Figure 1.1　Architecture of Paper

新点，提出了动态函数融合、资源解耦配置以及基于输入特征的动态资源配置方法，旨在优化无服务器工作流的性能与成本效益，并说明了论文的章节安排。

第二章相关技术背景介绍。本章首先介绍了无服务器计算的基础技术，包括容器技术、事件驱动架构、微服务架构以及资源管理和自动扩展技术，并概述了主流无服

务器计算平台的现状。接着，阐述了无服务器工作流的基本概念、工作流类型（顺序函数链和嵌套函数链）以及基于 AWS Lambda 的定价模型，并详细讨论了函数融合技术的原理与应用。最后，本章介绍了图算法理论（如 DAG 和关键路径算法）和随机森林模型的基本原理，为后续章节的研究提供了理论基础和技术支持。

第三章面向嵌套工作流的动态函数融合框架。本章针对无服务器嵌套工作流中的双重计费、级联冷启动和远程调用开销等问题，提出了一种动态函数融合框架。首先，通过实验验证了函数融合在嵌套工作流中的有效性，展示了其在减少成本和端到端时延方面的显著优势。接着，设计了一个反馈驱动的动态函数融合框架，包括函数融合器、动态函数融合路径搜索器和基础设施优化器，详细阐述了其工作原理和实现细节。最后，通过实验验证了该框架在多个典型嵌套工作流中的性能提升和成本节约效果。实验结果表明，该框架能够显著减少双重计费、级联冷启动和远程调用开销，并在成本和端到端时延优化方面优于基线方法，为无服务器嵌套工作流的优化提供了有效的解决方案

第四章基于资源解耦的工作流动态资源配置框架。本章针对无服务器工作流中的资源配置问题，提出了一种基于资源解耦的动态资源配置框架。首先，通过实验验证了资源解耦在不同工作流中的有效性，并分析了输入特性对资源需求的影响。接着，设计了一个基于资源解耦的工作流动态资源配置框架，包括基于关键路径的工作流采样调度模块、基于优先级调度的配置解耦搜索模块和针对输入敏感型工作流的动态配置模块。最后，通过实验验证了该框架在满足 SLO 的同时，显著降低了工作流的运行成本。实验结果表明，该框架在处理输入敏感型工作流时表现出更高的资源利用效率，为无服务器工作流的资源管理提供了有效的解决方案。

第五章总结与展望。本章首先总结了全文的研究成果，包括针对嵌套工作流的动态函数融合方法、基于资源解耦的自动配置方法以及面向输入敏感型工作流的动态资源配置方法的设计与实现。针对现有工作，对未来的研究方向进行了展望，包括拓展异构资源解耦维度以及结合函数融合与资源解耦配置以进一步提升性能与成本效益。

第 2 章 相关技术背景介绍

为了深入理解无服务器工作流在实际部署中面临的低效执行、资源捆绑配置以及输入敏感型工作流的动态资源配置这些挑战并探索解决方案，本章将系统介绍无服务器计算的基础知识、无服务器工作流的类型与定价模型，以及函数融合、图算法和随机森林模型等关键技术。这些技术为本文的研究提供了理论基础和方法支持，并为后续章节的深入分析和优化奠定了重要基础。

2.1 无服务器计算基础

无服务器计算作为一种新兴的云计算范式，近年来在学术界和工业界得到了广泛关注和应用。其核心思想是通过抽象底层基础设施，使开发者能够专注于业务逻辑的实现，而无需管理服务器的配置和维护。本节将系统介绍无服务器计算的底层技术，包括容器技术、事件驱动架构、微服务架构以及资源管理和自动扩展技术，以及无服务器计算平台的现状。这些技术为无服务器计算的高效性、灵活性和成本效益提供了重要支撑，并为后续章节的深入分析和优化奠定了理论基础。

2.1.1 无服务器计算底层技术

无服务器计算的实现依赖于一系列关键的底层技术，这些技术共同支撑起了无服务器架构的高效性、灵活性和成本效益。

容器技术 容器技术是无服务器计算的重要组成部分。借助容器化技术，应用程序及其依赖项可以被整合为一个轻量且可移植的单元，支持跨环境的无缝部署和迁移。常见的容器技术包括 Docker[59]和 Kubernetes[60]。

Docker 的核心原理如图2.1所示。作为开源的容器化平台，Docker 通过将应用及其依赖项整合到轻量级、可移植的容器中，支持应用在跨环境中的无缝迁移和一致运行。Docker 的底层技术主要依赖于容器镜像、Docker Daemon、容器运行时（containerd）、命名空间和控制组等机制。容器镜像是一种轻量化的打包格式，将应用程序及其依赖项整合在一起，通过联合文件系统实现镜像的分层存储。Docker Daemon（dockerd）是 Docker 的核心后台服务，负责管理 Docker 的整个生命周期，包括镜像、容器、网络和存储等。containerd 负责管理容器的生命周期，包括创建、启动、停止和删除容器。当 containerd 接收到创建容器的请求时，它会调用 containerd-shim，由 containerd-shim

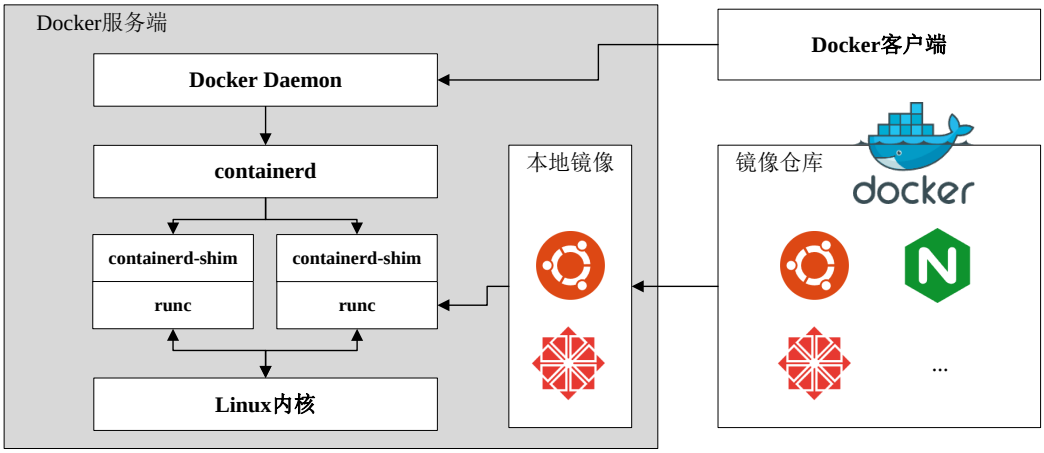


图 2.1　Docker 核心架构图
Figure 2.1　Core Architecture of Docker

负责启动实际的容器进程。containerd-shim 作为容器的实际管理进程，通过 runc 等工具与 Linux 内核交互，利用 Linux 内核的命名空间实现进程、网络、文件系统等的隔离，并通过控制组（cgroups）限制和监控容器的资源使用。

Docker 的核心原理如图2.2所示。Kubernetes 是一个开源的容器编排平台，专注于在多节点环境中管理、调度和编排容器化应用。Kubernetes 的底层技术主要依赖于 Pod、节点（Node）、控制平面（Control Plane）、调度器（Scheduler）和容器运行时（如 containerd 或 CRI-O）等核心组件。Pod 是 Kubernetes 的最小调度单元，为容器提供了一个共享的运行环境，它是一个或多个容器的集合，共享相同的网络命名空间和存储卷。节点是 Kubernetes 集群中的工作机器，每个节点上运行着 kubelet，它是 Kubernetes 的节点代理，负责与控制平面通信并管理节点上的 Pod 和容器。控制平面是 Kubernetes 集群的大脑，负责管理集群的整体状态，运行在 Master 节点上，其核心组件有提供 Kubernetes 的 REST API 的 API Server、保存集群的所有配置数据和状态的 etcd、负责将 Pod 调度到合适的节点上运行的 Scheduler 和运行副本控制器和节点控制器的 Controller Manager 等。容器运行时（如 containerd 或 CRI-O）是 Kubernetes 的底层组件，负责在节点上运行容器。Kubernetes 通过 CRI（Container Runtime Interface）与容器运行时交互，实现容器的创建、启动、停止和删除。当 Kubernetes 接收到创建 Pod 的请求时，API Server 会将其写入 etcd，随后调度器会根据资源情况将 Pod 分配到合适的节点。目标节点上的 kubelet 会通过 CRI 调用容器运行时，由容器运行时负责启动和管理容器。

事件驱动架构　事件驱动架构（Event-Driven Architecture, EDA）是无服务器计算

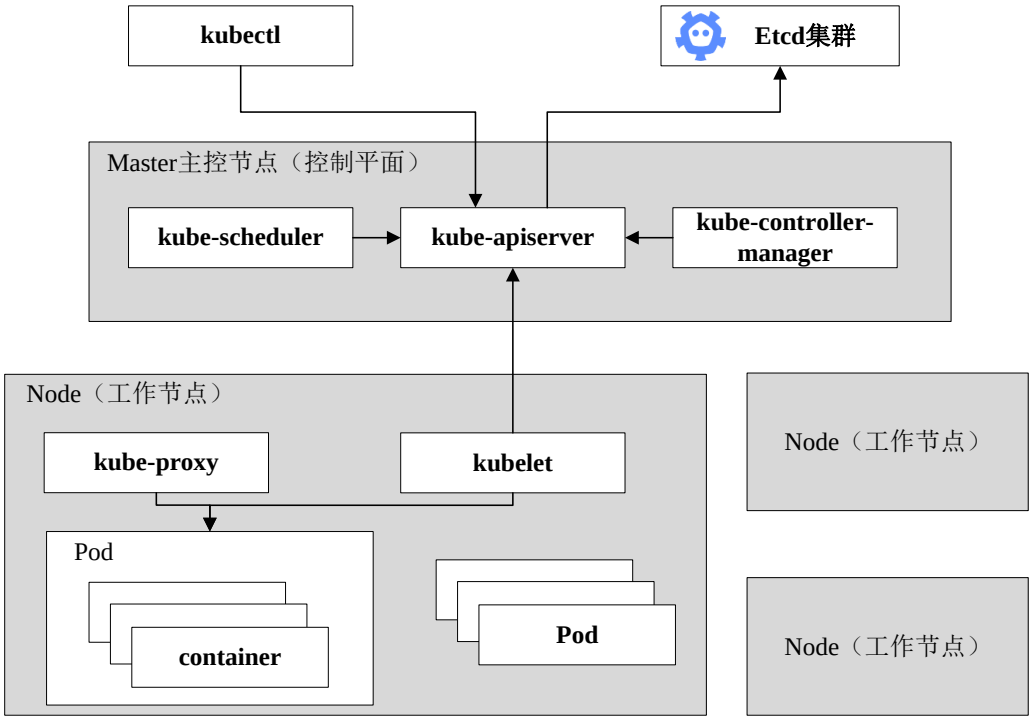


图 2.2 Kubernetes 核心架构图
Figure 2.2 Core Architecture of Kubernetes

的核心理念之一，在这种架构中，应用的执行是基于特定事件的触发，而不是持续运行。事件可以是多种形式，例如 HTTP 请求、数据库更改、消息队列中的消息等，当某个事件发生时，无服务器函数会被自动触发并执行相应的业务逻辑，这种基于事件的模型使得应用仅在需要时运行，从而高效利用资源。事件驱动架构的优势在于其高响应性和可扩展性，由于应用只在事件发生时运行，系统资源的使用率得到了极大优化。例如，在一个电商平台上，当用户下单时，系统会触发一个无服务器函数来处理订单。这个函数只在订单生成时运行，其他时间则处于空闲状态，从而节省了大量资源。此外，事件驱动架构还支持异步处理，使得应用能够处理大量并发事件，提高了系统的吞吐量。

微服务架构 无服务器计算受到了微服务架构的启发，将复杂的应用程序拆解为多个独立的功能服务单元，每个单元都可以单独部署、动态扩展和独立管理。这些小服务之间通过 API 进行通信，从而实现高可用、可扩展和易于维护的系统。在微服务架构中，应用被分解为一系列小型、独立的服务，每个服务执行单一的业务功能，这种架构使得应用可以灵活开发和扩展，适应快速变化的业务需求。在无服务器计算环境中，微服务架构的优势尤为明显。每个微服务可以独立部署和扩展，无需考虑整个

应用的复杂性，这种灵活性使得系统能够快速响应业务需求的变化，提高了开发效率。微服务架构还促进了代码的模块化和重用，每个微服务可以由不同的团队独立开发和维护，使用不同的技术栈，这种松耦合的设计使得系统更加健壮，易于维护。

资源管理和自动扩展 无服务器平台利用先进的资源管理和自动扩展技术，根据应用负载动态调整资源分配，确保应用在高负载下仍能高效运行，同时优化成本。在无服务器计算环境中，资源管理的关键在于自动化和智能化。无服务器平台会实时监控应用的负载情况，并根据预设的策略自动调整资源分配，例如，当一个无服务器函数被频繁调用时，平台会自动增加该函数的实例数量，以应对高并发请求。相反，当负载下降时，平台会自动减少实例数量，释放不必要的资源。这种动态调整机制不仅提高了系统的响应速度，还显著降低了资源浪费。自动扩展技术进一步增强了无服务器计算的弹性，无服务器平台可以根据应用的实际需求，自动扩展或缩减计算资源。无服务器平台还支持多租户环境，使得多个无服务器函数可以在同一物理或虚拟服务器上安全隔离运行，这种隔离性不仅提高了资源利用率，还增强了系统的安全性。

2.1.2　无服务器计算平台现状

无服务器计算模型自推出以来，各大云服务提供商纷纷推出了自己的商用无服务器计算平台，主要包括 AWS Lambda、Google Cloud Functions 和 Azure Functions 等。这些平台为无服务器应用程序的执行提供了强大的基础设施，但它们需要以特定的方式组合功能代码，并部署在云服务供应商的云平台上，从而导致长期的供应商锁定。为了克服这些限制，近年来出现了一些开源无服务器计算框架，如 Apache OpenWhisk[61]、Knative[62]等，这些框架可以部署在私有云环境上，也可以部署在边缘/雾环境的设备上，从而使无服务器计算能力具有更好的灵活性。

1. AWS Lambda

AWS Lambda 是亚马逊推出的无服务器计算服务，允许开发者运行代码而无需管理服务器。Lambda 支持多种编程语言，如 Java、Python、Go 和 Node.js 等流行的编程语言。该平台通过将 Amazon Simple Storage Service (S3) 的存储桶或 Amazon DynamoDB 表的数据变化、Amazon API Gateway 发起的 HTTP 请求以及使用 AWS SDK 进行的 API 调用作为事件触发器，实现了高效的事件驱动编程模型。

AWS Lambda 利用一种名为 Heartbeat 的虚拟化技术来执行函数请求，这是一种轻量级的微虚拟机（micro VM），为虚拟机内存利用率的最大化提供了优化。Lambda 采用特定的算法将虚拟机中的实例作为一个二进制打包问题来处理。AWS 提供了每月 100 万个请求和 40 万 GB-秒的计算时间的免费计算资源，超过此限制，函数执行

表 2.1 无服务器平台特性对比表
Table 2.1 Comparison Table of Serverless Platform Features

特性 \ 平台	AWS Lambda	Google Cloud Functions	Azure Functions	Apache OpenWhisk	Knative
是否开源	否	否	否	是	是
特有功能	S3、DynamoDB 等	与 Google Cloud 服务紧密集成	提供 Visual Studio 开发环境	动作、触发器与规则的编程模型	简化 Kubernetes，支持 Serverless
架构	部署于 AWS 平台	部署于 Google Cloud 平台	部署于 Azure 平台	支持云环境或本地部署	支持云环境或本地部署
并发实例管理和扩展	并发限制，按需自动扩展	按需自动扩展	按需自动扩展	按需自动扩展	按需自动扩展
定价模型	按请求次数、请求时间、内存配置计费	内存配置、CPU 配置分开计费	按执行期间消耗的平均内存计费	无	无

将按分配的内存和执行请求的数量收费。AWS Lambda 允许开发人员在部署阶段指定函数将访问的最大内存限制，并按允许的内存限制比例分配 CPU 功率。

2. Google Cloud Functions

Google Cloud Functions 是谷歌推出的一种无服务器计算服务，允许开发者在谷歌云运行独立的小段代码，而无需管理服务器或其他基础设施。这种服务基于事件驱动模型，意味着代码只在特定事件发生时执行，例如数据上传至存储桶、消息到达 Pub/Sub 主题，或者定时触发器激活。Google Cloud Functions 运行在谷歌的全球基础设施上，能够自动扩展以满足流量需求，并提供高可用性和低延迟的服务。Cloud Functions 可以与谷歌的其他云服务无缝集成，如 Cloud Storage、BigQuery 和 Firestore 等，方便开发者构建复杂的云应用。

Google Cloud Functions 的底层虚拟化技术依赖于 Google 的基础设施和容器技术。虽然 Google 并没有公开详细说明其虚拟化技术的具体名称，但可以了解到它是基于容器和轻量级执行环境的，这与 Kubernetes 的运作方式相似。在 Google Cloud Functions 中，每个函数都被封装在一个独立的容器中，这些容器是由 Google 的容器引擎管理的。这些容器可以在需要时快速启动，并在不需要时自动销毁，从而实现按需资源分配和成本效益。

3. Azure Functions

Azure Functions 是微软推出的无服务器计算服务，它紧密集成于 Azure 云平台，充分利用了微软的生态系统优势。这项服务的特性强调了与微软平台的深度整合，为开发者提供了无缝的开发体验和强大的功能支持。Azure Functions 可以与 Azure 的其他服务如 Azure Cosmos DB、Azure Storage、Azure Active Directory 等无缝集成，使得开发者能够利用这些服务构建复杂的企业级应用。

作为微软的产品，微软提供了专门的 Visual Studio Code 插件，为开发者提供了一个熟悉的开发环境，支持调试、测试和部署函数，极大地提高了开发效率。Azure Functions 还对 PowerShell 和 C# 语言提供了原生的支持，这对于使用这些语言的开发者来说是一个巨大的优势。Azure Functions 内置了与 Azure Monitor 和 Application Insights 的集成，提供了强大的监控和分析能力，帮助开发者实时跟踪应用的性能和健康状况。

4. Apache OpenWhisk

Apache OpenWhisk 是一个开源的无服务器计算平台，最初由 IBM 发起，后成为 Apache 软件基金会的一部分。它允许开发者在无需管理服务器的情况下运行代码片段，响应各种事件，非常适合构建事件驱动的应用程序。OpenWhisk 支持多种编程语言，包括 Node.js、Python、Java、Swift 和 PHP，并提供自动扩展能力，根据事件负载自动调整执行实例。此外，它能够轻松集成外部服务和 API，拥有活跃的社区支持和丰富的文档资源。

工作原理上，OpenWhisk 基于事件驱动模型，当预定义事件发生时，事件代理检测到事件并将其路由到一个或多个动作（函数）。这些函数在沙箱环境中执行，执行结果随后返回给请求者或传递给下一个动作。OpenWhisk 的架构包括控制器、调度器、执行者、存储、事件代理和 API 网关。控制器管理用户请求和权限验证，调度器分配动作执行，执行者运行函数代码，存储管理元数据和临时数据，事件代理监听事件源，API 网关提供 HTTP 接口以触发函数执行。

5. Knative

Knative 是一个开源的无服务器计算框架，由 Google 发起并被 Cloud Native Computing Foundation (CNCF) 接管，旨在简化在 Kubernetes 上构建、部署和管理无服务器应用程序的过程。它提供了一系列强大的特性，包括事件驱动架构、自动扩展、标准化的构建和部署流程、跨平台的可移植性、高级的路由和流量管理功能，以及作为无服务器和容器化桥梁的能力。这些特性使开发者能够构建响应各种事件的应用程序，

并根据流量和负载自动调整资源，同时利用 Kubernetes 的强大功能和灵活性。

Knative 的工作原理依赖于几个核心组件，包括 Build、Serve 和 Eventing。Build 负责将源代码转换为容器镜像，Serve 管理网络路由和流量，确保服务的高可用性和可扩展性，而 Eventing 支持事件驱动架构，允许服务响应外部事件。这些组件共同实现了自动扩展、流量管理和事件响应，确保了应用程序能够根据实际需求动态调整资源。此外，Knative 还集成了 Istio 服务网格和监控工具，如 Prometheus 和 Jaeger，以提供高级流量管理和性能分析。

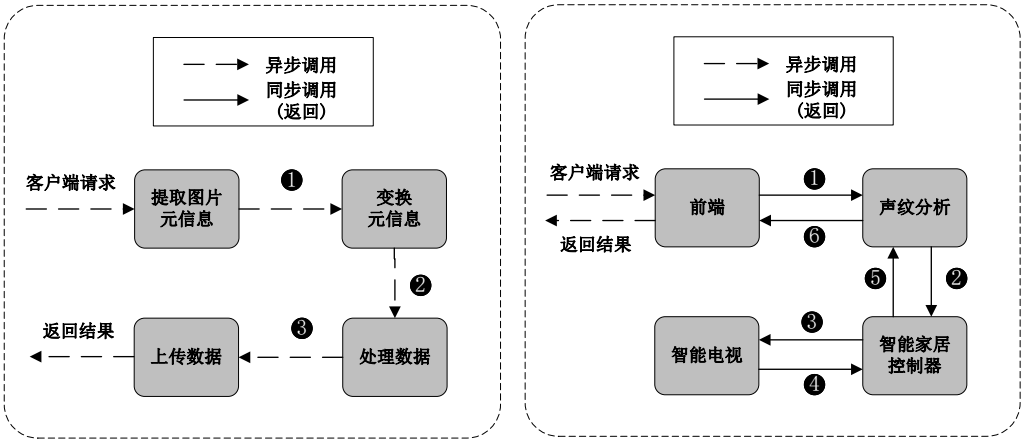
2.2 无服务器工作流

无服务器工作流将复杂的业务逻辑分解为多个独立的函数，并通过特定的组合方式实现任务的协同执行。工作流的设计和管理直接影响着系统的性能、成本和可扩展性。在无服务器计算环境中，工作流的组织方式主要包括顺序函数链和嵌套函数链两种类型，每种方式都有其独特的优势和适用场景。此外，无服务器工作流的定价模型和函数融合技术也是优化性能和成本的关键因素。本节将详细介绍无服务器工作流的类型、定价模型以及函数融合技术，为后续章节中针对工作流低效执行和资源配置问题的研究提供理论基础和技术支持。

2.2.1 工作流类型

在无服务器计算领域，工作流的组织和管理是实现高效、可扩展应用程序的关键。无服务器平台通常支持两种主要的工作流组合方法：顺序函数链（Sequential Function Chaining）和嵌套函数链（Nested Function Chaining）。

顺序函数链示例如图2.3a所示，其依赖于平台或第三方提供的协调机制，如 AWS Step Functions 和 Azure Durable Functions，这些机制负责管理工作流，并在交互函数之间传递中间状态。顺序函数链的主要特点包括协调机制、中间状态传递和冷启动分摊。协调机制是指由平台或第三方提供专门的协调服务，用于管理和编排函数链的执行。例如，AWS Step Functions 使用状态机来定义和执行工作流，Azure Durable Functions 则通过持久化函数状态来实现协调。函数之间的中间状态通过协调机制传递，确保数据的一致性和完整性。这种机制允许函数链中的每个函数在前一个函数完成后立即执行，无需等待外部触发。由于协调机制的存在，冷启动开销可以被分摊到整个链中的多个函数上，从而减少单个函数的冷启动延迟，这对于需要长时间运行的工作流特别有利。此外，顺序函数链还提供了强大的错误处理和重试机制，确保工作



(a) 顺序函数链示例　　(b) 嵌套函数链示例

图 2.3　不同组合方法的无服务器工作流示例
Figure 2.3　Examples of Serverless Workflows with Different Combination Methods

流的可靠性和稳定性。

嵌套函数链示例如图2.3b所示，其通过平台提供的 SDK 或直接网络请求调用其他函数，这种方法更为广泛支持，因为任何允许函数进行网络请求的无服务器平台都能实现嵌套函数链。嵌套函数链无需额外的协调机制，实现更为灵活。开发者可以根据需要自由组合和调用函数，无需依赖特定的协调工具。然而，嵌套函数链面临更高的超时风险，尤其是在冷启动情况下，如果链中的某个函数执行时间过长，可能会导致整个链的超时。此外，嵌套函数链的调试和监控相对复杂，因为每个函数都是独立调用的，难以追踪整个链的执行状态。尽管如此，嵌套函数链在处理复杂的业务逻辑和动态工作流时具有显著优势，因为它允许开发者根据实际需求灵活调整函数调用顺序和逻辑。

顺序函数链通过协调器提供强大的状态管理和性能优化能力，适用于需要顺序执行和可靠性的场景。顺序函数链的冷启动开销可以被分摊，适合需要长时间运行的工作流，并且提供了强大的错误处理和重试机制，确保工作流的可靠性和稳定性。而嵌套函数链则提供了极高的灵活性和独立性，适用于复杂的业务逻辑和场景。嵌套函数链允许开发者根据实际需求灵活调整函数调用顺序和逻辑，但面临更高的超时风险和调试监控复杂性。顺序函数链更适合需要中间状态传递和冷启动分摊的场景，而嵌套函数链则更适合需要灵活组合和调用函数的场景。

2.2.2 定价模型

在无服务器计算环境中，定价模型是服务提供商和用户之间经济交互的核心。考虑到市场占有率，本研究的工作流定价模型均基于 AWS Lambda 的定价模型[63]，该模型主要通过函数的内存配置、执行时间和请求次数来计算费用。AWS 在每月开始时提供一些免费请求和转换，以及超大量请求后的价格优惠策略，本文的重点不在于探讨无服务器供应商的定价策略，故本文的定价模型仅考虑免费请求消耗后的基础价格，认为此种情况符合大多数开发者的计费情况。

AWS Lambda 的定价模型中，函数的费用主要由其内存配置和执行时间决定。具体来说，费用是根据函数在执行期间所消耗的内存量（以 GB 为单位）和执行时间（以毫秒为单位）的乘积来计算的。公式如下：

$$\text{Cost}_{\text{memory}} = \text{Memory (GB)} \times \text{Duration (ms)} \times \text{Price}_{\text{GB-ms}} \tag{2-1}$$

其中，$\text{Price}_{\text{GB-ms}}$ 是 AWS Lambda 设定的固定费率。根据 AWS 提供的计费手册，$\text{Price}_{\text{GB-s}}$ 为 0.00001667 美元，也就是 AWS Lambda 计价中的基础固定费率。

除了内存配置和执行时间外，函数的请求次数也是定价模型中的一个重要因素。AWS Lambda 对每百万次请求收取固定费用。公式如下：

$$\text{Cost}_{\text{requests}} = \text{Requests} \times \text{Price}_{\text{million requests}} \tag{2-2}$$

根据 AWS 提供的计费手册，$\text{Price}_{\text{million requests}}$ 为 0.20 美元，也就是 AWS Lambda 计价中的基础固定次数费用。

综合考虑内存配置、执行时间和请求次数，整体定价模型可以表示为：

$$\text{Cost} = \text{Cost}_{\text{memory}} + \text{Cost}_{\text{requests}} \tag{2-3}$$

其中，$\text{Cost}_{\text{memory}}$ 代表内存配置与执行时间产生的费用，$\text{Cost}_{\text{requests}}$ 代表请求次数产生的费用。

2.2.3 函数融合

在无服务器计算环境中，函数融合（Function Fusion）是一种优化技术，旨在减少同步调用函数间的延迟和资源开销，如图2.4所示。

首先定义以下术语：

- **任务（Task）**：开发者编写的软件函数，即在函数融合前由开发者所编写的细粒度的业务逻辑，表示为 T_i，其中 i 是任务的唯一标识符。

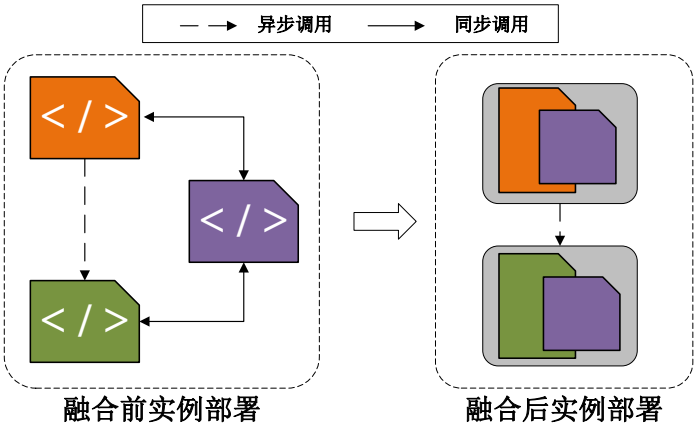


图 2.4　函数融合示例
Figure 2.4　Function Fusion Example

- **函数（Function）**：实际执行的部署构件，可认为是部署的容器实例，引入了函数融合后可包含一个或多个任务，表示为 F_j，其中 j 是函数的唯一标识符。

每个函数 F_j 包含一个融合组（Fusion Group），即作为该特定函数一部分执行的一组任务。融合组可以表示为 G_k，其中 k 是融合组的唯一标识符。

调用同一融合组中的任务可以与编译器中的内联相比较，即其他任务的代码直接在该函数内部执行。具体来说，如果任务 T_i 和 T_j 属于同一个融合组 G_k，则 T_i 调用 T_j 的执行方式为：

$$\text{Exec}(T_i, T_j) = \text{Inline} \tag{2-4}$$

当调用另一个融合组中的任务时，执行将移交给负责该组任务的函数。具体来说，如果任务 T_i 属于融合组 G_k，而任务 T_j 属于融合组 G_l，且 $G_k \neq G_l$，则 T_i 调用 T_j 的执行方式为：

$$\text{Exec}(T_i, T_j) = \text{Remote} \tag{2-5}$$

为了部署应用程序，需要知道所有融合组、函数的基础设施配置以及如何将调用移交给其他融合组。本文将此信息称为融合设置（Fusion Setup），表示为 $\mathcal{F}$：

$$\mathcal{F} = \{(F_j, G_k) \mid F_j \text{ 包含融合组} G_k\} \tag{2-6}$$

本文使用简短的符号来描述融合组：位于融合组中的任务用括号括起来，用逗号分隔。不同的融合组用连字符分隔。例如，开发者可能指定了三个任务 A、B 和 C，

其中 A 调用 B 和 C。使用融合组 $(A,B)-(C)$ 为例，代表任务 A 和 B 在同一个融合组中，任务 C 在自己的组中。这意味着 A 通过内联任务 B 的代码本地调用 B，而 A 对 C 的调用则移交给另一个函数。这种表示法是简化的，无法显示所有可能的融合组，因为融合组是有向图。例如，任务 A 可能内联任务 B，但反之则不然。

2.3 图算法理论

图算法是计算机科学和数学中的重要研究领域，广泛应用于任务调度、网络分析、路径规划以及工作流优化等场景。在无服务器计算中，图算法为管理和优化复杂的工作流提供了强大的工具，尤其是通过有向无环图（DAG）和关键路径算法（CPM）等技术，能够有效表示任务之间的依赖关系并优化执行顺序。本节将系统介绍图论的基础知识，包括图的表示方法、有向无环图（DAG）的定义与特性，以及关键路径算法的原理与应用。这些算法为无服务器工作流的任务调度和性能优化提供了理论基础，并为后续章节中针对工作流低效执行和资源配置问题的研究奠定了重要基础。

2.3.1 图论基础

图论是数学的一个分支，研究由节点（顶点）和边（连接节点的线）组成的图结构。图论在计算机科学、网络分析、任务调度等领域有广泛应用。

图 G 是一个有序对 (V,E)，其中 V 是节点的集合，表示图中的顶点；E 是边的集合，表示顶点之间的连接关系。每条边 $e \in E$ 可以表示为一个有序对 (u,v)，其中 $u,v \in V$，表示顶点 u 和 v 之间存在一条边。子图 $G' = (V',E')$ 是图 $G = (V,E)$ 的子集，其中 $V' \subseteq V$ 且 $E' \subseteq E$。子图保留了原图中的一部分顶点和边。

图可以分为有向图和无向图。有向图中的边具有方向性，表示从一个顶点到另一个顶点的单向连接，可以表示为 $G = (V,E)$，其中 $E \subseteq V \times V$。无向图中的边没有方向性，表示两个顶点之间的双向连接，可以表示为 $G = (V,E)$，其中 $E \subseteq \{\{u,v\} | u,v \in V\}$。

路径和环是图中的重要概念。路径是一个顶点序列 $v_1, v_2, ..., v_k$，使得对于所有 $1 \leq i \leq k$，存在边 $(v_i, v_{i+1} \in E$。路径的长度定义为路径中边的数量。环是一个路径 $v_1, v_2, ..., v_k$，使得 $v_1 = v_k$ 且 $k > 1$，表示从一个顶点出发经过若干条边后可以回到该顶点。连通性和强连通性描述了图的连通性。在无向图的结构中，连通性的判定标准在于图中任意两个顶点之间均需存在至少一条路径以使其相互可达；而对于有向图而言，强连通性的定义则更为严格，其要求图中任意两个顶点之间不仅需要存在从起点到终点的路径，还需同时存在从终点到起点的路径，即双向可达性必须得以满足。

度描述了顶点的连接关系。在无向图的语境下，某一顶点的度可被定义为与该顶点直接关联的边的总计数量；而在有向图的情形中，顶点的度则需进一步区分为入度与出度两个维度，其中入度表征的是以该顶点为终点的边的数目，而出度则表征的是以该顶点为起点的边的数目。邻接矩阵和邻接表是图的常见表示方法。邻接矩阵是一个 $|V| \times |V|$ 的矩阵 A，其中 $A[i][j] = 1$ 表示顶点 i 和顶点 j 之间存在一条边，否则 $A[i][j] = 0$。邻接表是一个数组，数组的每个元素是一个链表，表示与该顶点相连的所有顶点。

2.3.2　有向无环图

无论是顺序函数链还是嵌套函数链，都可以用有向无环图（DAG, Directed Acyclic Graph）来表示。DAG 是一种图结构，广泛应用于计算机科学和工程领域，用于表示任务之间的依赖关系和执行顺序，其中节点表示任务或函数，边表示任务之间的依赖关系。顺序函数链可以看作是线性 DAG，其中每个节点依赖于前一个节点；而嵌套函数链则可以表示为更复杂的 DAG，其中节点之间可能存在多重依赖关系。DAG 的主要特点包括有向性、无环性和并行性，这些特性使得 DAG 成为管理和优化复杂工作流的理想工具。

在 DAG 中，节点（Vertex）表示任务或函数，边（Edge）表示任务之间的依赖关系。具体来说，DAG 可以定义为一个有序对 $G = (V, E)$，其中 V 是节点的集合，表示任务或函数，是有向边的集合；E 表示任务之间的依赖关系，每条边 $E \subseteq E$ 可以表示为一个有序对 (u, v)，其中 $u, v \in V$，表示顶点 u 和 v 之间存在一条边。例如，如果任务 A 完成后才能执行任务 B，则存在从 A 到 B 的有向边。这种有向性确保了任务可以按照依赖关系顺序执行，不会出现死锁或无限循环。此外，DAG 中不存在环路，即不存在任务之间的循环依赖，这进一步确保了任务可以按照正确的顺序执行。

无环性可以通过以下公式来定义：对于任意节点 $v \subseteq V$，不存在一个序列 $v_1, v_2, ..., v_k$，使得 $v_1 = v_k$ 且 $(v_i, v_{i+1}) \subseteq E$ 对于所有 $1 < i < k$ 成立。换句话说，DAG 中不存在一个节点序列，使得从某个节点出发经过若干条边后可以回到该节点。

DAG 的一个重要特性是拓扑排序（Topological Sorting），即对节点进行排序，使得所有依赖关系都得到满足。DAG 的另一个重要特性是并行性。在 DAG 中，只要任务之间没有依赖关系，它们可以并行执行，从而提高执行效率。如果任务 A 和任务 B 没有依赖关系，它们可以并行执行，而不需要等待对方完成。这种并行性在处理大规模数据处理和复杂计算任务时尤为重要，可以显著缩短任务的执行时间。

DAG 还提供了强大的依赖管理机制，可以清晰地表示任务之间的依赖关系，确

保任务按照正确的顺序执行。通过 DAG，开发者可以定义任务之间的依赖关系，并利用并行执行来提高工作流的效率。在无服务器计算中，DAG 可以用于表示复杂的工作流，无论是顺序函数链还是嵌套函数链。

2.3.3 关键路径算法

关键路径算法（Critical Path Method, CPM）是一种广泛应用于项目管理的图算法，主要用于任务调度和工作流优化。该算法通过构建一个有向无环图（DAG）来表示任务及其依赖关系，其中节点代表任务，边表示任务之间的依赖关系，并附加权重以表示任务的执行时间。关键路径算法的伪代码如算法2.1所示。

算法 2.1 关键路径算法

Data: 图 $G = (V, E)$，权重函数 $w : E \to \mathbb{R}^+$
Result: 关键路径 critical_path，总工期 duration
Initialize: EST $\leftarrow$ array of size $|V|$ LST $\leftarrow$ array of size $|V|$
for ***each*** $v \in V$ **do**
  EST[v] $\leftarrow 0$ LST[v] $\leftarrow \infty$
end
LST[end] $\leftarrow$ EST[end] // 终止节点的最晚开始时间等于最早开始时间
for ***each*** $v \in V$ *in topological order* **do**
  for ***each*** $u \in predecessors(v)$ **do**
    EST[v] $\leftarrow \max(\text{EST}[v], \text{EST}[u] + w(u, v))$
  end
end
for ***each*** $v \in V$ *in reverse topological order* **do**
  for ***each*** $w \in successors(v)$ **do**
    LST[v] $\leftarrow \min(\text{LST}[v], \text{LST}[w] - w(v, w))$
  end
end
for ***each*** $v \in V$ **do**
  if $EST[v] = LST[v]$ **then**
    critical_path.$append(v)$
  end
end
duration $\leftarrow$ EST[end]
Return critical_path, duration

首先，从起始节点开始计算每个节点的最早开始时间（Earliest Start Time, EST），即在不违反依赖关系的前提下任务可以开始执行的最早时间，起始节点的 EST 为 0，其他节点 v 的 EST 为所有前驱节点 u 的 EST 加上边 (u, v) 的权重中的最大值。接着，从终止节点开始计算每个节点的最晚开始时间（Latest Start Time, LST），即在不延迟项目总工期的情况下任务可以开始执行的最晚时间，终止节点的 LST 等于其 EST，其

他节点 v 的 LST 为所有后继节点 w 的 LST 减去边 (v, w) 的权重中的最小值。关键路径上的节点满足 EST 等于 LST，这条路径是从起始节点到终止节点的最长路径，决定了项目的总工期。

在无服务器计算中，关键路径算法可以用于优化工作流和任务调度。通过分析任务之间的依赖关系，确定关键路径，可以有效地管理任务的执行顺序和时间，避免任务延迟和资源浪费。例如，在顺序函数链和嵌套函数链中，关键路径算法可以帮助确定哪些任务是关键任务，从而优先处理这些任务，确保整个工作流的顺利执行。

2.4　随机森林模型

随机森林（Random Forest）作为一种强大的集成学习（Ensemble Learning）算法，近年来在机器学习和数据分析领域得到了广泛应用。其核心思想是通过集成多个决策树模型来构建一个强学习器，利用投票或平均的方式综合各树的预测结果，从而显著提升模型的准确性和鲁棒性。随机森林不仅能够处理高维数据，还能有效应对噪声和过拟合问题，因此在分类和回归任务中表现出色。在无服务器计算环境中，随机森林模型可以用于预测函数的运行时间、内存消耗等性能指标，为资源优化和动态配置提供重要支持。本节将详细介绍随机森林的基本原理、构建过程及其在无服务器计算中的应用，为后续章节中基于输入特征的动态资源配置研究奠定理论基础。

决策树作为一种基础的机器学习模型，其核心思想是通过递归的方式将数据集逐步划分为若干子集，使得每个子集内的样本尽可能趋于同质化。决策树的构建过程通常涵盖三个关键步骤：特征选择、数据划分以及递归构建。其中，特征选择旨在从所有候选特征中筛选出最优特征作为当前节点的划分依据，常用的评价指标包括信息增益（Information Gain）和基尼不纯度（Gini Impurity）等。数据划分是基于所选特征将数据集分割为多个子集，每个子集对应一个分支。递归构建则是通过对当前数据空间的每一次子集划分，迭代地执行特征选择与数据划分，直至达到某种预先定义的收敛准则或停止阈值，从而确保整个构建过程在满足特定约束条件的前提下，逐步逼近全局最优解或局部均衡状态。

决策树的数学表示可以表示为一个递归的划分过程，假设数据集 D 包含 n 个样本，每个样本有 m 个特征。决策树的划分过程可以表示为：

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \tag{2-7}$$

其中 x_i 是特征向量，y_i 是目标变量。

随机森林的核心思想是通过引入随机性来构建多个决策树，并将这些树的结果进行集成。具体步骤如下：

1. **Bootstrap 采样**：在数据重构的初始阶段，对原始数据集实施一种基于概率置换的抽样策略，允许样本在抽取过程中以可重复的方式被重新引入，从而生成一系列经过重采样的子数据集（通常称为 Bootstrap 样本）。这些子数据集在规模上与源数据集保持同构性，然而由于抽样过程的随机性与可重复性，某些数据点可能会在单一子数据集中呈现出多重共现的现象，而另一些数据点则可能因概率分布的随机波动而被暂时排除在特定子数据集之外，进而形成一种动态的、非对称的数据分布格局。
2. **随机特征选择**：在每一棵决策树的构造过程中引入一种基于随机子空间采样的特征筛选机制，从全局特征集中动态抽取一个限定规模的子集作为当前节点的候选特征池。这种策略不仅能够有效削弱特征间潜在的共线性干扰，还能在模型集成的层面上通过引入特征空间的异质性，促进学习器之间的多样性生成，从而在整体上提升模型的泛化能力与鲁棒性。
3. **决策树构建**：基于前述 Bootstrap 采样技术所生成的动态子数据集，并结合随机特征子空间的筛选机制，系统性地构建一系列决策树模型。每一棵决策树在其生长过程中，均以最大化局部信息增益为目标，通过递归地分割数据空间，逐步扩展其分支结构，直至达到某种预定义的收敛准则或停止阈值，从而确保每棵树在其独立的特征空间与数据分布中，充分挖掘潜在的分类或回归规律。
4. **集成预测**：在分类任务的场景下，随机森林模型的最终输出是通过集成所有基决策树的个体预测结果，统计各树预测类别的众数来确定最终的类别标签；而在回归任务的框架中，模型的预测值则是对所有决策树的输出进行一种基于算术平均的聚合操作，从而在数值上收敛到一个能够综合反映各树预测趋势的均衡点，以此实现对目标变量的稳健估计。

假设构建了 B 棵决策树，每棵树的预测结果为 $f_b(x)$，其中 $b = 1, 2, \ldots, B$。最终的预测结果 $\hat{y}$ 是所有决策树投票的结果：

$$\hat{y} = \text{argmax}_c \sum_{b=1}^{B} \mathbb{I}(f_b(x) = c) \tag{2-8}$$

其中 $\mathbb{I}(\cdot)$ 是指示函数，c 是类别标签。

最终的预测结果 $\hat{y}$ 是所有决策树预测结果的平均值：

$$\hat{y} = \frac{1}{B}\sum_{b=1}^{B} f_b(x) \tag{2-9}$$

随机森林模型在预测无服务器函数的运行时间和内存等性能指标方面具有显著优势。无服务器函数通常运行在高度动态和异构的环境中，其性能受到多种因素的影响，如输入数据的大小、函数的复杂度、运行时环境等。随机森林通过构建多棵决策树，能够有效地捕捉这些非线性关系，从而提高预测的准确性。

2.5　本章小结

本章首先通过对无服务器计算底层技术的分析，阐述了容器技术、事件驱动架构等底层技术如何支撑无服务器计算，并介绍了常见平台。其次详细探讨了无服务器工作流的两种主要类型——顺序函数链和嵌套函数链，介绍了基于 AWS Lambda 的定价模型，还讨论了函数融合技术，为无服务器工作流的性能提升提供了理论支持。在图算法理论部分，重点介绍了有向无环图（DAG）和关键路径算法，这些算法为任务调度和工作流优化提供了重要工具。最后，本章详细阐述了随机森林模型的基本原理及其在无服务器计算中的应用，为基于输入特征的动态资源配置研究奠定了理论基础。

第 3 章　面向嵌套工作流的动态函数融合框架

随着无服务器计算的普及，FaaS 在构建复杂应用时面临嵌套工作流执行效率低下的问题，如双重计费、级联冷启动和远程调用开销等。本章提出一种动态函数融合框架，旨在通过将多个函数融合为单个执行单元，优化嵌套工作流的性能和成本效率。本章首先分析无服务器嵌套工作流部署中的低效执行问题，通过初步实验验证函数融合的有效性，然后介绍动态函数融合框架的设计与实现，最后通过实验评估框架性能。

3.1　研究动机

本节将围绕无服务器嵌套工作流部署中的低效执行问题展开讨论，分析其背后的原因，并通过实验初步验证函数融合在解决这些问题中的有效性。

3.1.1　无服务器嵌套工作流部署的低效执行

尽管 FaaS 提供了强大的抽象和灵活性，商业 FaaS 平台的技术和定价模型在某些方面仍然滞后，云服务供应商仅针对包含单个无服务器函数的应用程序进行了优化。开发者编写的细粒度软件函数与部署和实例化在云中的可执行构件之间的一对一映射，往往会导致次优的性能和成本效率。在实际应用中，无服务器工作流，尤其是嵌套工作流的部署面临诸多挑战。

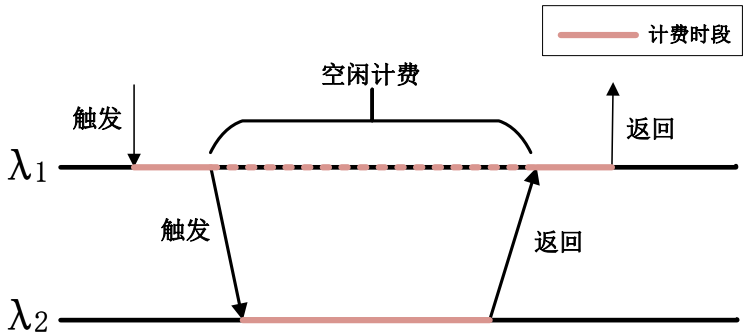


图 3.1　双重计费
Figure 3.1　Double Pricing

嵌套工作流中的双重计费问题尤为突出，当一个 FaaS 函数对另一个函数进行同步调用时，即它等待该调用的结果，应用程序的执行时间会被计费两次。如图3.1所

示，被调用的函数会产生成本，然而等待的函数也会被计费，尽管它没有执行任何有用的工作。尽管函数之间的同步调用模式对于构建大型、复杂的应用程序至关重要，但从成本角度来看，双重计费的问题可能是不可接受的。

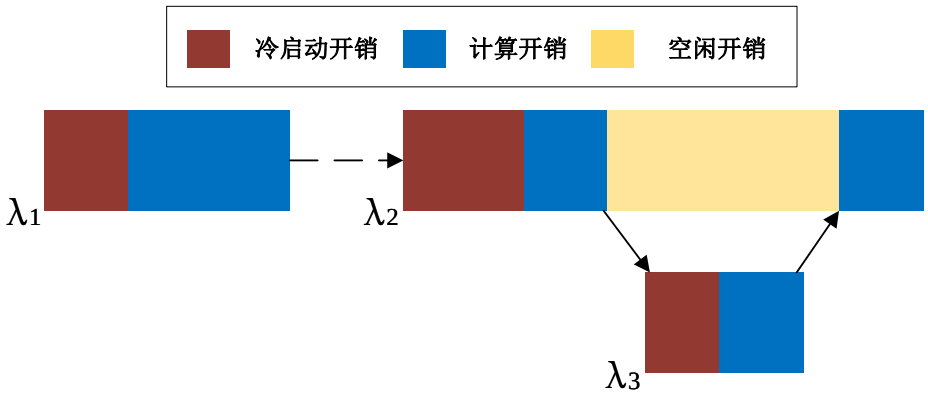


图 3.2　级联冷启动
Figure 3.2　Cascading Cold Start

级联冷启动问题也是无服务器嵌套工作流部署中的一个重要挑战。如图3.2所示，在按顺序执行的函数链组成的 FaaS 组合中，冷启动开销会累积，因为链中的每个函数实例仅在调用时才会被实例化。这不仅增加了应用程序的端到端延迟，还显著提高了其执行成本。

复杂工作流函数间的通信开销也会加重无服务器嵌套工作流的低效执行问题。FaaS 平台是为通过网络进行函数调用而构建的，主要使用 HTTP 请求。对于外部客户端，通过 HTTP 路由器访问函数是有意义的，但它为内部函数调用创建了相当大的调用开销。Grambow[64]等人发现，商业 FaaS 平台中从函数调用函数的网络传输时间大约为 50 毫秒，这对于运行时间较短的函数序列来说是一个显著的开销。在按毫秒计费的 FaaS 定价模型下，这不仅会导致应用程序的端到端延迟增加，还会增加执行成本。

3.1.2　函数融合有效性初步验证

许多的工作发现，通过将工作流函数融合，可以最大限度地解决上述问题。对于一个嵌套工作流，融合前后可以显著减少双重计费、冷启动开销和远程调用开销等问题。具体来说，通过融合函数，容器实例在调用同步函数时可以将资源分配给被调函数，主调函数只需要等待回调即可，从而减少不必要的同步调用，避免双重计费。除此之外，对函数进行融合还能减少函数实例的创建次数，降低级联冷启动开销，减

少远程调用的次数，降低网络传输时间，从而提高整体性能和成本效益。

为了验证对嵌套工作流的函数进行融合的有效性，本章选择了一个实际的无服务器应用场景——Serverless TrainTicket[65]，这是复旦大学软件工程实验室（Fudan SE Lab）开发的一款基于无服务器架构的火车票查询系统，架构如图3.3所示。该系统旨在展示如何利用无服务器技术构建实时、弹性的在线服务，并通过真实的业务场景帮助开发者更好地理解无服务器技术的应用。此外，Serverless TrainTicket 还是一个用于无服务器基准测试的系统，使其成为验证函数融合效果的理想平台。

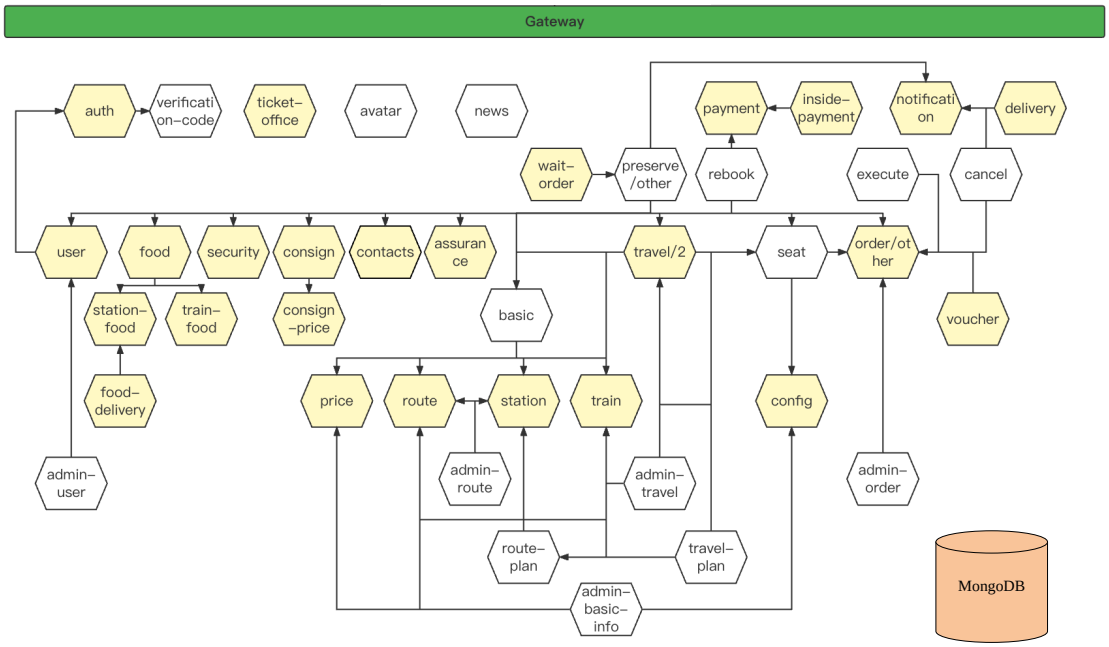
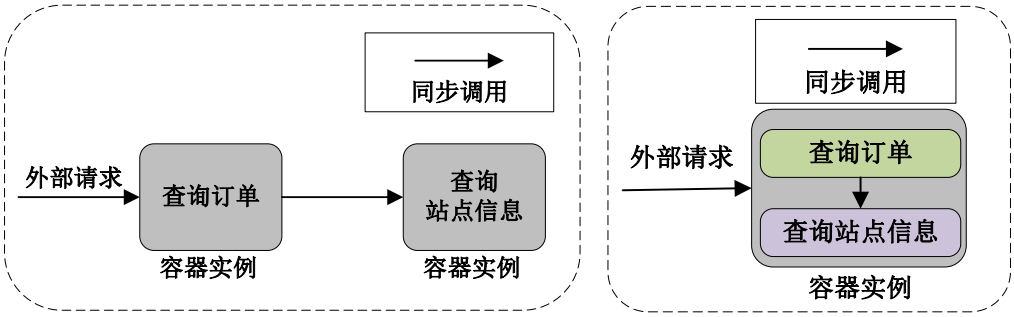


图 3.3　Serverless TrainTicket 架构图
Figure 3.3　Architecture of Serverless TrainTicket

在 Serverless TrainTicket 中，订单查询工作流是一个典型的嵌套工作流示例。该工作流由两个核心无服务器函数组成：查询订单函数和查询站点信息函数。查询订单函数作为工作流的入口，负责接收前端通过 HTTP 请求传递的用户订单查询信息，并从 MongoDB 数据库中检索相关的订单详情。随后，查询订单函数通过同步调用的方式调用查询站点信息函数，以补充订单中缺失的站点信息，最终将完整的订单信息返回给前端。这种嵌套调用的设计在无服务器架构中非常常见，但也带来了双重计费、级联冷启动和远程调用开销等问题。

为了验证函数融合的效果，对该订单查询工作流进行了实验。首先，对工作流中的两个函数进行了改造，以支持函数融合的操作。具体来说，查询订单函数在调用查询站点信息函数时，会优先检查本地是否存在对应的 Jar 包函数可供调用。如果存在，

则直接调用本地函数；否则，再通过网关进行远程调用。



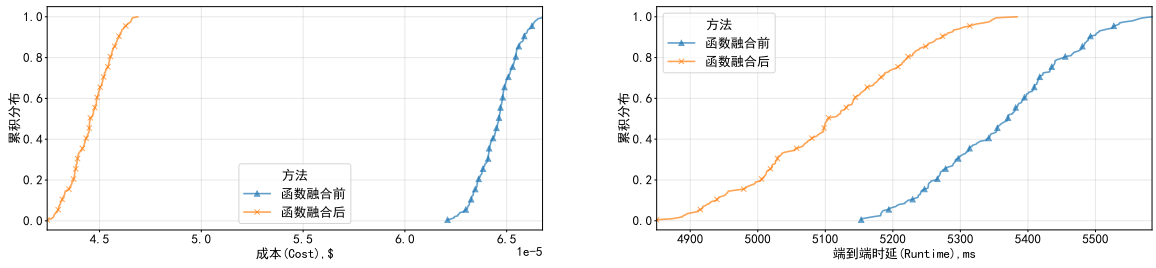
(a) 融合前的工作流
(a) Workflow before Fusion

(b) 融合后的工作流
(b) Workflow after Fusion

图 3.4　订单查询工作流融合前后的流程图
Figure 3.4　Flowchart of the Order Inquiry Workflow Before and After Fusion

在实验中分别部署了融合前和融合后的工作流，如图3.4所示。对于融合前的工作流，独立部署了查询订单函数和查询站点信息函数的实例，并通过网关接收函数调用请求。而对于融合后的工作流，将查询站点信息的 Jar 包函数与查询订单函数绑定在同一个实例中进行部署，从而实现了函数的本地调用。

为了模拟真实的用户请求场景，向融合前和融合后的工作流分别发送了 200 次请求。在每次请求中，记录了工作流的开销、端到端时延，以初步评估函数融合的效果。



(a) 工作流函数融合前后的开销
(a) Cost Overhead of Workflow Functions Before and After Fusion

(b) 工作流函数融合前后的端到端时延
(b) End-to-End Latency of Workflow Functions Before and After Fusion

图 3.5　工作流函数融合前后的性能对比
Figure 3.5　Performance Comparison of Workflow Functions Before and After Fusion

实验结果如图3.5所示。图3.5a展示了工作流函数融合前后的开销对比，结果表明，融合前工作流的平均每次开销为 6.45×10^5\$，融合后工作流的平均每次开销为

4.46×10^5\$，融合后工作流的平均每次开销降低了 30.9%。这一结果主要归因于函数融合减少了同步调用期间空闲的容器实例，进而显著降低了双重计费所带来的开销。图3.5b展示了工作流函数融合前后的端到端时延对比，结果显示，融合前工作流的平均每次端到端时延为 5361.85ms，融合后工作流的平均每次开销为 5110.48ms，融合后工作流的平均每次端到端时延降低了 4.7%。这一改进主要得益于减少了通过网关进行 HTTP 请求的时延，以及级联冷启动的时延。

综上所述，认为通过减少远程调用、避免双重计费以及降低级联冷启动开销，函数融合有效解决了无服务器嵌套工作流部署中的关键问题，能够在在嵌套工作流中带来显著的性能和成本效益提升。基于上述实验结果，将在本章节中进一步构建一个通用的无服务器函数融合框架，以支持普适性的嵌套工作流优化，为开发者提供一种高效、灵活的解决方案。

3.2 面向嵌套工作流的动态函数融合框架架构

本节将围绕动态函数融合框架的设计展开讨论，阐述其设计目标、总体架构以及核心组件的工作流程。

3.2.1 设计目标

本研究旨在构建一个面向嵌套工作流的动态函数融合框架，该框架能够根据实际运行时的监控数据，自主地优化函数融合设置，以满足开发者在成本和性能方面的需求。

现有的方法往往依赖于开发者手动配置和优化，这不仅增加了开发者的负担，还可能导致配置不当，影响应用程序的性能和成本效益。本框架致力于提供一个简单的编程模型，使开发者能够专注于业务逻辑的实现，而无需担心底层优化细节。通过提供通用的同步调用库和要求的被调编程模型，开发者只需按照规范开发被调函数，并使用指定的库进行同步调用。系统将自动处理所有优化过程，从而减轻开发者的负担，提高开发效率。这种方法的好处在于，开发者可以专注于核心业务逻辑，而无需花费大量时间和精力在底层优化上，从而显著提升开发效率和产品质量。

其次，许多优化方法依赖于对应用程序行为的建模，这需要大量的测试和分析数据，且难以应对实际运行中的动态变化。本框架通过捕获系统在实际负载下的行为，避免对应用程序行为的建模，从而专注于实际调用情况。这种方法不仅简化了系统架构，还减少了不必要的复杂性。通过直接分析实际运行数据，系统能够更准确地识别

和优化关键路径，从而提高整体性能。这种方法的优势在于，它能够适应实际运行中的动态变化，避免了对应用程序行为的过度依赖。通过专注于实际调用情况，系统能够更有效地优化常见场景，忽略罕见的边缘情况，从而为平均应用程序运行找到更好的解决方案。

3.2.2　总体架构

本框架的核心是一个反馈驱动的自主部署系统，它通过收集和分析 FaaS 监控数据，迭代优化函数融合设置。这种优化设置不仅取决于开发者的偏好（如成本和性能目标），还受到运行时效应的影响。框架重点是使应用程序在实际负载下能够进行迭代优化，同时保持对开发者的额外负担最小化。开发者只需遵循编程模型，其他所有优化过程都将自动处理，无需用户建模或对应用程序的工作方式做出任何假设。架构不进行任何应用程序行为的建模，而是仅捕获系统在实际负载下的行为。对于某些场景，可能不如一些依赖测试运行或分析数据的更重模型的方法有效，因为必须依赖准确的监控数据。本质上，只能考虑实际中实际看到的调用。然而，这也可以是一个优势，因为它忽略了罕见的边缘情况，因此可以为平均应用程序运行找到更好的解决方案。

如图3.6所示，本系统包含三个主要组件：函数融合器、动态函数融合路径搜索器和基础设施优化器。

函数融合器负责记录函数之间的调度请求、执行信息，以及对函数进行融合与部署操作。它为开发者提供了通用的同步调用库和要求的被调编程模型。当开发者按照要求开发被调函数，并使用指定的库进行同步调用后，融合器能够在之后的融合操作中进行各种自动化的融合操作，无需开发者介入。此外，它还会记录无服务器工作流的实际执行信息，如调度请求、执行时间、冷启动时间等数据，提供给函数融合路径搜索器使用。

动态函数融合路径搜索器通过检索监控数据来推导应用程序的调用图，并用执行信息（如延迟值）对其进行注释。然后，它使用可扩展的优化策略模块来推导改进的融合设置。对融合后的设置进行持续监测，达到验收条件（如端到端成本降低）后，将此融合变更为永久形式。若未达到退出融合路径搜索的条件，则继续搜索下一个融合路径。在下一小节中描述了一个启发式方法。当融合搜索结束，进入基础设施优化阶段。

基础设施优化器基于机器学习中流行的梯度下降优化算法。其思想是通过在每次迭代中选择向最小成本方向（左或右方向）来找到某个指标的最小值，直到达到最

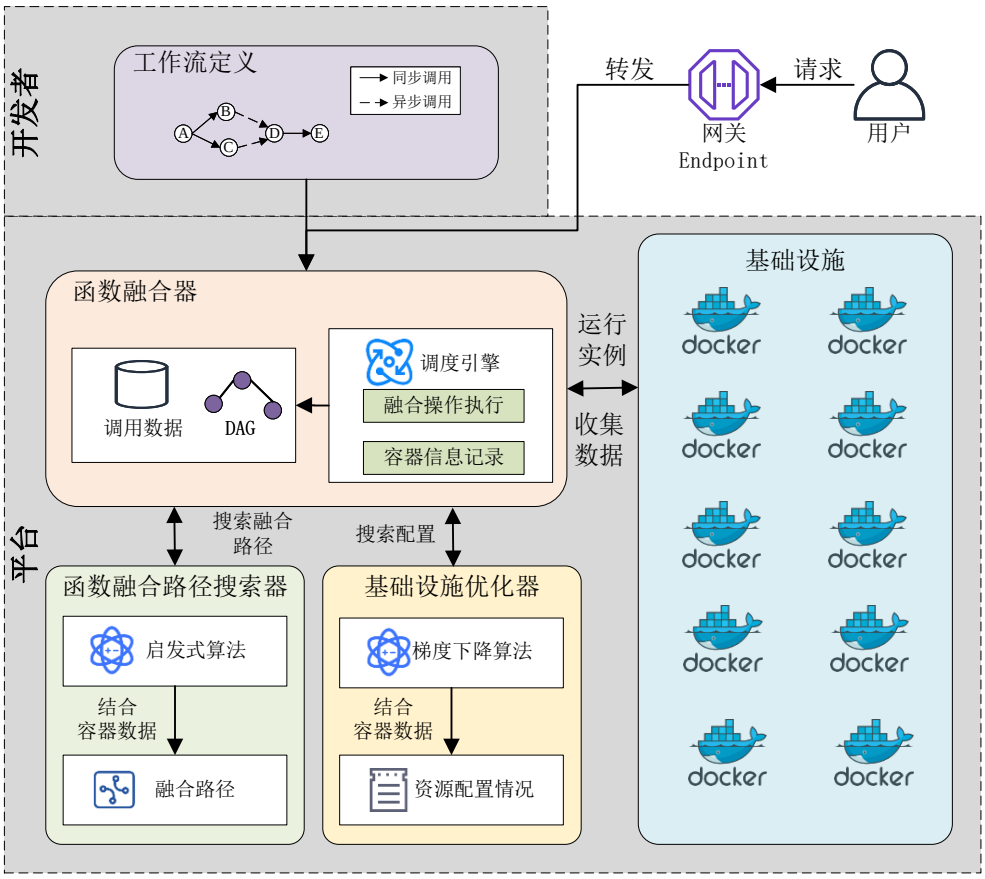


图 3.6　函数融合框架架构图

Figure 3.6　Architecture Diagram of Function Fusion Framework

小值。从提供的内存列表中随机选择一个内存值，计算其成本指标及其左侧邻居的成本。如果邻居的成本高于当前点的成本，算法继续在当前点的右侧执行（在成本递减的方向上，否则在左侧）。如果当前成本小于最小成本，则更新最小成本。

3.3　面向嵌套工作流的动态函数融合框架实现

本节将详细介绍面向嵌套工作流的动态函数融合框架的实现细节，重点阐述其核心组件——无服务器函数融合器、动态函数融合路径搜索器以及基础设施优化器的工作原理与设计思路。

3.3.1　无服务器函数融合器

为了实现无服务器函数融合器，本框架为开发者提供了同步调用库及编程规范。同步调用库如图3.7所示，以 Java 库的形式提供，旨在简化开发者对无服务器函数的

同步调用过程。通过引入该库，开发者可以轻松实现函数的本地或远程调用，而无需关心底层通信细节。编程规范只涉及被调的同步函数，以追求尽可能少的侵入性编程，为开发者快速入手融合器带来便利。

```
public class SyncCallLibrary {
    // 对外远程调用接口
    public static String SyncCall(String functionName,
                                  Map<String, Object> params)
    public static Map<String, Object> ParseParams(String[] args)
    public static String WrapResultAsJSON(Map<String, Object> result)
    // 内部实现
    private static Map<String, Object> callLocalFunction(…)
    private static Map<String, Object> callRemoteFunction(…)
}
```

图 3.7　同步调用库定义
Figure 3.7　Definition of Synchronous Invocation Library

同步调用的入参包括 functionName 和 params 两个部分：functionName 是注册函数的名称，类型为字符串，该名称用于标识需要调用的函数，确保调用的唯一性和准确性。params 是传入参数，类型为 `Map<String, Object>`。其中，键为参数名，值为参数值。

同步调用的返回结果是一个 `Map<String, Object>`，其中键为结果名，值为结果值。返回结果以 JSON 格式打包，便于跨平台和跨语言的解析与使用。

该库的同步调用函数，首先尝试调用本地工作目录下的 JAR 包中的函数，如果本地 JAR 包中存在该函数，则直接调用并返回结果，这种方式确保了在本地环境中快速、高效地执行函数调用。如果本地 JAR 包中不存在该函数，则通过网关进行远程函数调用；远程调用通过 HTTP 请求实现，请求体中包含函数名和传入参数。远程调用返回的结果同样以 JSON 格式打包，并解析为 `Map<String, Object>` 返回给调用者，这种一致的结果格式，简化了调用者的后续处理逻辑。

被调函数的编程模型遵循如图3.8所示，分为以下几个步骤：

- **参数读取**：被调函数从命令行中读取参数，并进行解析。这种方式确保了函数在不同环境下的可移植性和兼容性。该方法无需开发者实现，同步库里已有封装函数，直接引入即可。

```
import SyncCallLibrary;
public class CalledFunction {
    public static void  main(String[] args) {
        try {
          // 1. 参数读取：从命令行中读取参数并解析
          Map<String, Object>  Input = ParseParms(args);
          // 2. 业务逻辑执行：根据传入的参数执行业务逻辑
          // 3. 结果返回：将结果以JSON格式返回
          System.out.println(WrapResultAsJSON(result));
         } catch (Exception e) {…}
    }
}
```

图 3.8　同步调用被调函数编程模型
Figure 3.8　Programming Model for Synchronously Invoked Callee Functions

- **业务逻辑执行**：被调函数执行相应的业务逻辑。这代表着开发者原先所开发的无服务器业务逻辑，需要开发者根据业务场景自行开发。
- **结果返回**：被调函数通过调用同步调用库的函数，将结果以 JSON 格式返回，并输出到控制台中。注意，只有调用同步库里的封装函数，才能确保 JSON 格式的结果被主调函数正确解析。

函数融合器是无服务器函数融合系统的核心组件，负责记录所有无服务器函数调用的请求信息、执行信息等。调用请求信息，指函数名、传入参数、调用时间等；执行信息，包括执行时间、返回结果、错误信息等，这些信息为后续的分析和优化提供了基础数据。在融合路径搜索器请求时，函数融合器会提取内部储存的工作流的调用 DAG 图，将执行信息记录在 DAG 的节点上，发送给融合路径搜索器供其使用。这种方式确保了执行信息的完整性和可追溯性。收到融合路径搜索器的函数融合路径后，函数融合器将融合的两个函数的 Java 程序打包成 JAR 程序，放入同一工作目录下，生成 Dockerfile 文件，再生成镜像。接着，修改工作流的调用 DAG 图，以确保新的请求到来时会按照执行融合后的工作流形式执行。当函数融合路径搜索结束后，会进一步启动基础设施优化器。最后，当基础设施优化结束后，会启动一个守护线程，该线程会定时触发函数融合、基础设施优化，以便根据在线数据反馈定期优化工作流部署，以取得最佳性能。

函数融合器的工作流程如图3.9所示：

1. **工作流上传**：用户上传工作流，函数融合器接收工作流，并为其构造 DAG 图，

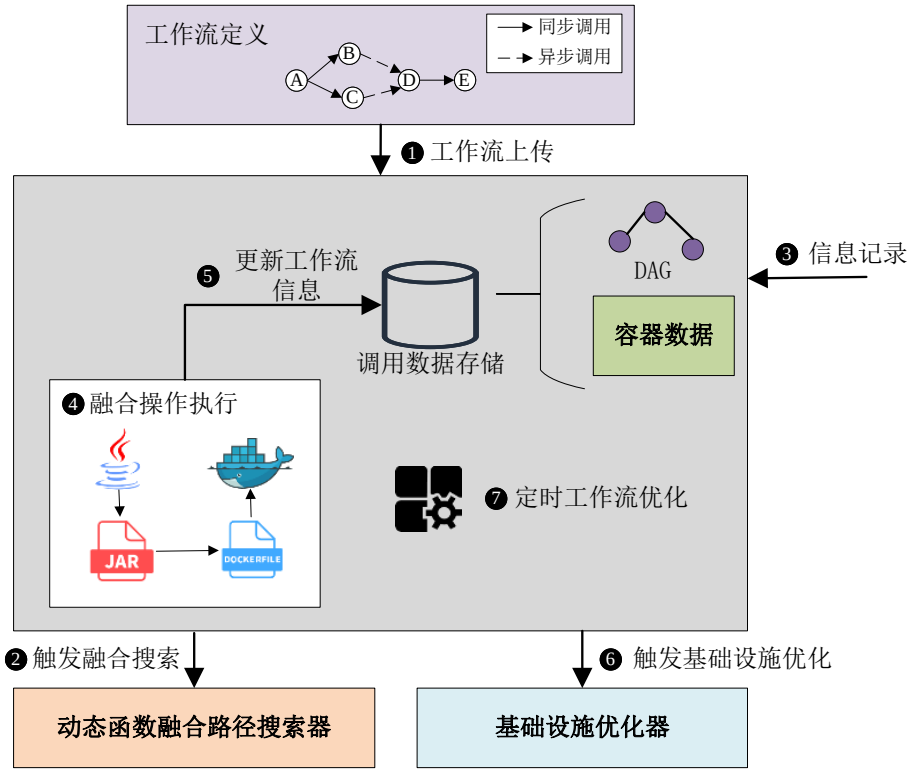


图 3.9　函数融合器
Figure 3.9　Function Fusion Engine

分配储存调用信息的空间。

2. **触发融合搜索**：函数融合器为工作流触发动态函数融合路径搜索器，由搜索器使用在线数据持续优化工作流部署。
3. **信息记录**：函数融合器记录工作流所关联的函数的请求信息和执行信息。
4. **融合操作执行**：收到融合路径搜索器的函数融合操作后，函数融合器执行以下步骤：
   4.1. 将融合的 N 个 Java 程序打包成 JAR 程序。
   4.2. 将 JAR 程序放入同一工作目录下。
   4.3. 生成 Dockerfile 文件，定义镜像的构建过程。
   4.4. 生成镜像。
5. **更新工作流信息**：函数融合器清空原工作流部署信息，生成新的调用 DAG 图。
6. **触发基础设施优化**：收到融合路径搜索器的函数融合路径搜索结束的请求后，函数融合器会触发基础设施优化器，以进一步优化工作流部署。
7. **定时工作流优化**：收到基础设施优化器优化结束的请求后，函数融合器会启动

守护线程。

3.3.2 基于启发式算法的动态函数融合路径搜索器

在本节中提出了一组规则，这些规则允许创建一个好的（但不一定是最佳的）融合设置。为此，本框架的启发式方法在应用程序运行时基于监控数据分析其多个方面。

将任务融合到与调用任务相同的函数中执行对整体性能和成本的影响最大。考虑一个设置示例，其中一个任务调用另一个任务，并且必须在继续之前同步等待结果。如果被调用的任务在另一个融合组中，这将导致双重计费，同样，远程调用比本地调用花费的时间要长得多。因此，这两个任务可能受益于在同一个融合组中。如果一个任务是异步调用的，则该任务的远程执行不会导致双重计费，因为调用任务不需要等待被调用任务完成。虽然将函数简单地放入一个实例中，可以再进一步减少容器冷启动开销以及远程开销，但必须考虑到这违背了无服务器应用程序运行的准则，而且通常容器的配置有限，放入过多函数就严重影响实例调度。为了释放关键路径，异步任务应该移交给另一个函数，即放入另一个融合组。

这些一般规则在每种情况下可能不是最优的：调用另一个函数的开销可能超过本地执行任务所需的时间，在这种情况下，最好在本地运行它。即使在冷启动期间，对于运行时间较长的任务也可能是如此，以防止级联冷启动。

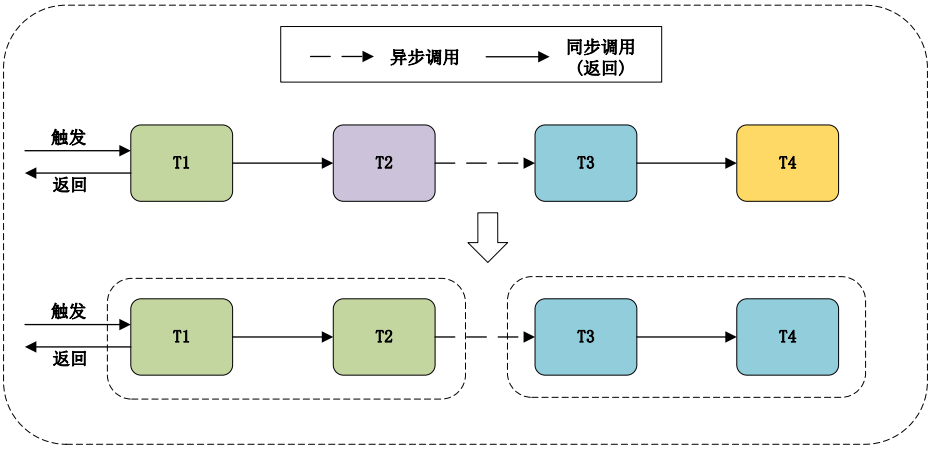


图 3.10 函数融合路径实例
Figure 3.10 Function Fusion Path Example

路径优化的一个示例如图3.10所示：T1 接收请求，向 T2 发出同步调用，T2 向 T3 发出异步调用（例如，用于日志记录目的），最后 T3 向 T4 发出同步调用。路径优化

算法 3.1 函数融合启发式算法

```
Input: 工作流的 DAG 图 G
Output: 优化后的融合组集合 F
// 步骤 1：划分融合组，不同的融合组以异步调用为界限，同步调用的函数属于同一个融合组
F ← partition_dag(G);
// 步骤 2：动态收集数据
while 数据 < 阈值 do
    数据 ← collect_data(f);
    // 收集线上执行数据，直到数据满足阈值
// 步骤 3：并行开启融合路径搜索
foreach f ∈ F do
    longest_sync ← find_longest_sync(f);
    if longest_sync ≠ None then
        融合结果 ← fuse_functions(longest_sync);
// 步骤 4：触发函数融合器融合函数，部署工作流
deploy_workflow();
// 步骤 5：检查 SLO 和成本
foreach f ∈ F do
    if check_slo(f) = 违规 then
        回退融合组 ← f;
        mark_failure(f);
    else
        成本 ← calculate_cost(f);
        if 成本 > 原成本 then
            回退融合组 ← f;
            mark_failure(f);
// 步骤 6：重复直到没有融合组可进一步尝试融合
while 存在可融合的融合组 do
    // 重复步骤 2 到步骤 5
return F
```

后，所有同步任务都被融合，所有异步任务都被拆分，导致融合组 (T1,T2)-(T3,T4)。

提出的启发式算法如算法3.1所示，首先将一个工作流的 DAG 图，根据异步和同步调用情况，分为若干个融合组，具体规则是，不同的融合组以异步调用为界限，也就是说出现了同步调用的函数就是一个融合组。框架在融合组内部去进行函数融合的操作，因为框架的目标是为嵌套工作流去进行优化。在融合组内部，框架动态收集线上的执行数据，数据满足一定数量后，将开启融合路径搜索流程。认为在融合组

里同步调用时间最长的函数将带来最大的二重计费问题，框架将优先解决这个问题，也就是框架将优先尝试融合调用时长最长的同步调用的主调和被调函数。框架的融合是每个融合组并行进行的，这将带来效率的提升。然后，调用融合器进行融合、部署后，一段时间后向函数融合器请求工作流 DAG 的执行数据。如果 SLO 违规，则将端到端时延增加的融合组标记，这个融合组的尝试回退，以保证 SLO 不违规。同时，会标记这两个函数的融合失败，之后不会再尝试融合这两个函数；如果 SLO 不违规，则计算融合组的成本，将成本上升的融合组的操作回退，同理也会标记这两个函数的融合失败。如此往复，直到没有融合组可进一步尝试融合为止。

3.3.3 基于梯度下降算法的基础设施优化器

基础设施优化器用于根据定义的优化目标自动为无服务器工作流找到最佳内存配置，内置了基于机器学习中流行的梯度下降优化算法，算法流程如算法3.2所示，思想是继续通过在每次迭代中选择向最小成本方向（左或右方向）来找到某个指标的最小值，直到达到最小值。

该算法通过基于梯度下降的方法优化有向无环图（DAG）中各节点的内存配置。首先，算法初始化每个节点的步数计数器（第 1 行），用于记录每个节点在优化过程中遇到的局部最小值次数。同时，算法为每个节点维护成本和运行时间的记录（第 2-3 行），以便在优化过程中动态更新这些指标。

在每次迭代中，算法首先保存当前内存配置及其相关数据（第 5 行），包括当前节点的内存值、成本和运行时间。接下来，算法计算当前内存配置的左右邻居的内存值。左邻居的内存值通过从当前内存值减去内存步长得到，而右邻居的内存值则通过加上内存步长得到。如果左邻居或右邻居的内存值超出预设的内存范围（例如最小值 128MB 或最大值 1024MB），则将其限制在边界值内。在计算左右邻居的内存值后，算法判断是否需要重新运行工作流以评估这些邻居的成本和运行时间。如果某个邻居的内存值尚未被评估过，则需要重新运行工作流。具体来说，算法会更新 DAG 中所有节点的内存配置，设置新的内存限制，并调用工作流入口函数来执行工作流，同时记录运行日志。通过分析日志，算法更新 DAG 中每个节点的成本和运行时间。如果某个邻居的内存值已经被评估过，则直接使用已有的成本和运行时间记录（第 6-13 行）。

在获得左右邻居的成本和运行时间后，算法通过比较这些值来决定内存配置的移动方向（第 14-21 行）。具体来说，算法会检查左邻居和右邻居的成本是否低于当前节点的成本。如果左邻居的成本更低，则算法将内存配置向左移动；如果右邻居的

算法 3.2 梯度下降优化 DAG 内存配置

Input: *DAG*, 入口函数 ID, 内存步长, 最大迭代次数, 阈值计数, SLO 运行时间, 调用次数
Output: 优化后的 DAG 内存配置

```
初始化每个节点的步计数 step_counts ;
初始化每个节点的成本和运行时间记录 containers_cost, containers_runtime ;
计算并记录初始内存配置的成本和运行时间 ;
for 每次迭代 = 1 到最大迭代次数 do
    保存当前内存配置和相关数据 ;
    // 计算左邻和右邻的内存配置
    if 左邻需要运行 then
        更新 DAG 为左邻内存配置，运行并记录成本和运行时间 ;
    else
        使用已记录的左邻成本和运行时间 ;
    if 右邻需要运行 then
        更新 DAG 为右邻内存配置，运行并记录成本和运行时间 ;
    else
        使用已记录的右邻成本和运行时间 ;
    // 根据成本选择最优方向
    foreach 节点 node_id do
        if 步计数 < 阈值计数 then
            if 左邻成本最低 then
                向左移动内存配置 ;
            else if 右邻成本最低 then
                向右移动内存配置 ;
            else
                达到局部最小，增加步计数 ;
    // 更新局部最小点
    foreach 局部最小点 do
        向更小内存方向移动 ;
return 优化后的 DAG 内存配置 ;
```

成本更低，则向右移动。如果左右邻居的成本均不低于当前节点的成本，则认为当前节点达到了局部最小值，并增加该节点的步数计数。

为了克服梯度下降算法可能陷入局部最小值的问题，算法引入了步数计数器。当某个节点的步数计数达到预设的阈值计数时，算法认为该节点已经充分探索了其附近的内存配置，不再对其进行进一步优化。对于已经达到局部最小值的节点，算法会进一步向更小的内存方向调整配置，以探索更低的成本。

在每次迭代结束时，算法会检查是否达到了最大迭代次数或满足收敛条件。如果未达到最大迭代次数，则继续进行下一次迭代，重复上述过程。最终，算法输出优化后的 DAG 内存配置，以最小化成本并满足运行时间约束。

此外，为了进一步提高算法的鲁棒性，算法在达到局部最小值后，会随机选择未被探索过的内存配置，并重复梯度下降操作。这种随机选择机制确保了算法不会在遇到的第一个局部最小值处停止，而是继续探索其他可能的内存配置，直到达到阈值计数或无可用内存为止。通过这种方式，算法能够在全局范围内寻找更优的内存配置，从而提高优化效果。

3.4 实验分析

本节将通过实验验证面向嵌套工作流的动态函数融合框架的有效性。首先介绍了实验的软硬件环境，接着描述了两种基线方法——Costless 和 RightFusion，作为对比基准，以评估提出的框架在性能和成本优化方面的优势。随后设计了基于 Serverless TrainTicket 系统的实验，选取了三个典型的嵌套工作流作为测试用例，分别评估函数融合和基础设施优化的效果。最后，通过对实验结果的分析，验证了所提出框架在减少端到端时延、冷启动时间和执行成本方面的显著优势，为无服务器嵌套工作流的优化提供了有力的实践支持。

3.4.1 实验软硬件环境

实验在一台配备 4 个 Intel Xeon Gold 6248R 处理器（96 个物理核心）和 512GB 内存的机器上运行。工作流在独立的 Docker 容器中执行，实现了各种融合情况下的函数实例执行。这种环境设置使得框架能够在不同的融合配置下运行工作流，从而评估不同融合情况对性能和成本的影响。

3.4.2 基线方法介绍

本文提出的动态函数融合框架将与两种基线方法进行比较：Costless 和 RightFusion。

Costless Costless 是一种基于资源优化和成本最小化的无服务器函数融合方法，其核心思想是通过构建函数成本图并使用约束满足问题（CSP）求解器来寻找满足 SLO 约束的最小成本融合方案。Costless 的执行流程主要包括以下几个步骤：

1. **Profile 阶段**：通过 128MB 内存执行函数，记录函数的最大内存需求，并向上取最近的可分配内存值（例如 128MB、256MB、320MB 等）作为实际的资源配

置。同时，记录调度时间和实际执行时间。

2. **DAG 转换为 FnSeq**：将工作流的 DAG 图转换为函数序列（FnSeq），采用广度优先搜索的方式进行遍历。
3. **融合方案生成与评估**：在论文中，Costless 通过构建函数成本图并使用 CSP 求解器来寻找最优融合方案。然而，在复现过程中，本章简化了这一步骤，直接对所有可能的融合方案（共 2^{N-1} 种）进行逐个评估，通过未融合前的工作流数据估计每个方案的时间和成本，并选择满足 SLO 约束的最小成本方案。

RightFusion RightFusion 是一种基于 SLO 约束的无服务器函数融合方法，其核心思想是通过资源优化和函数融合来满足每个函数的 SLO 要求，从而实现性能和成本的平衡。RightFusion 的前提条件包括为每个函数预定义 SLO 以及提供工作流的 DAG 表示。RightFusion 的执行流程主要包括以下几个步骤：

1. **Profile 阶段**：由于函数的执行时间与输入无关，RightFusion 不需要使用机器学习模型进行预测。相反，它直接对每个函数在几种 vCPU 分配情况下的执行时间进行 profile，并记录每种配置下的执行时间。然后，选择满足 SLO 要求的最小 vCPU 配置作为该函数的资源分配方案。
2. **函数融合**：基于 DAG 的结构，RightFusion 尝试将两两相邻的函数进行融合。融合的条件包括：该函数只有一个父节点；该父节点只有一个子节点；两函数的 vCPU 分配相同。

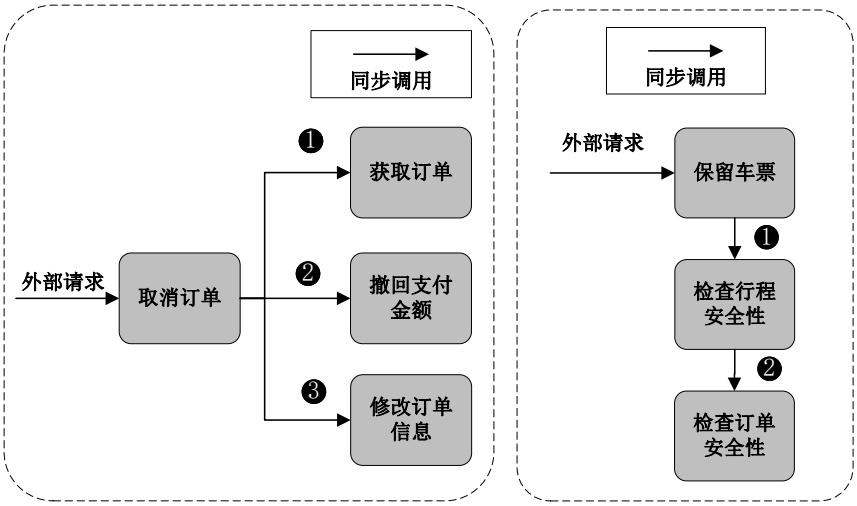
   如果满足上述条件，则将这两个函数融合为一个函数，从而减少调用开销和资源分配的复杂性。

3.4.3　实验介绍

为了全面评估动态函数融合框架与基线方法的性能优劣，从 Serverless TrainTicket 中提取了三个典型的嵌套工作流作为测试基准：取消订单、保留车票和获取行程剩余车票。这些工作流涵盖了不同的复杂度和嵌套层次，能够有效验证函数融合方法在实际应用中的表现。

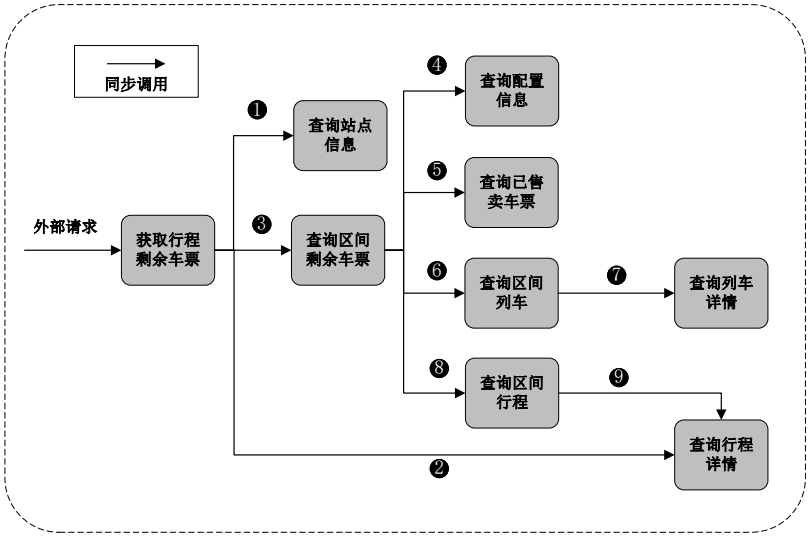
如图3.11所示，三个工作流的架构如下：

- **取消订单工作流**：包含 4 个函数，其中取消订单函数作为工作流的入口。该函数以同步调用的形式依次调用获取订单函数、撤回支付金额函数和修改订单信息函数。
- **保留车票工作流**：包含 3 个函数，其中保留车票函数作为工作流的入口。该函数以同步调用的形式调用检查行程安全性函数，而检查行程安全性函数再以同



(a) 取消订单工作流
(a) Cancel Order Workflow

(b) 保留车票工作流
(b) Reserve Ticket Workflow



(c) 获取行程剩余车票工作流
(c) Retrieve Remaining Tickets for Itinerary Workflow

图 3.11 三个典型嵌套工作流的架构图
Figure 3.11 Architecture of Three Typical Nested Workflows

步调用的形式调用检查订单安全性函数。

- **获取行程剩余车票工作流**：最为复杂，包含 9 个函数，以获取行程剩余车票函数作为工作流的入口。该工作流涉及多个嵌套调用，展示了嵌套工作流中函数调用的复杂性。

为了进行融合实验并分析测试结果，设计了以下实验步骤：首先，对每个融合

情况，发送 200 次用户请求，以模拟实际的用户操作，并记录每次请求的端到端时延、冷启动时间和开销，以全面评估工作流的性能。在未进行基础设施优化前，每个函数默认配置 512MB 的内存，为了简化优化空间，限制内存配置范围为 128MB 至 1024MB。采用暴力搜索方法，探索所有可能的函数融合组合，收集三个工作流的所有可能融合结果的成本，以评估函数融合结果与最优结果的接近性，并通过对比不同融合方案的成本，量化函数融合对性能和成本的影响。最后，在三个工作流上，分别运行了 Costless、RightFusion 以及所提出动态函数融合框架的算法，进行函数融合实验，通过对比这些方法的性能指标，评估不同方法在实际应用中的优劣。

在实验中，主要关注端到端时延、冷启动时间和开销这三个性能指标，分别反映了工作流的整体执行效率、响应速度和经济性。本实验的主要目标包括通过对比融合前后的性能指标，验证函数融合对减少端到端时延、冷启动时间和开销的效果，通过与 Costless 和 RightFusion 的对比，评估动态函数融合框架在性能和成本优化方面的优势。

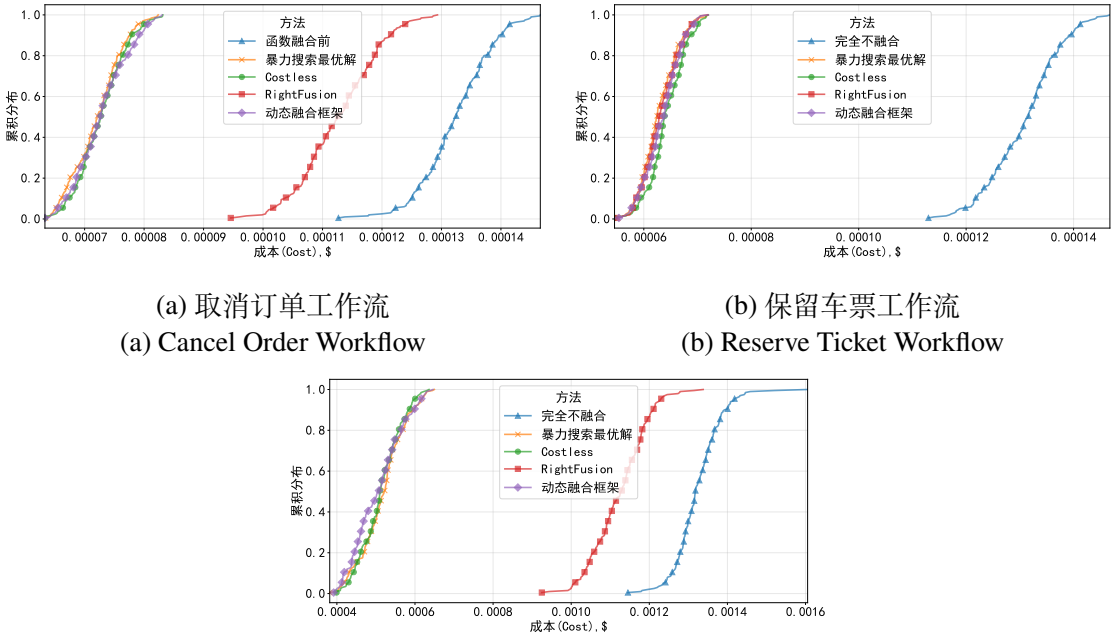
3.4.4　测试结果及分析

3.4.4.1　函数融合效果分析

为了全面评估函数融合在嵌套工作流中的实际效果，本节对三个典型工作流（取消订单、保留车票和获取行程剩余车票）进行了详细的实验分析。实验中分别使用了三种方法：本文所提出的动态融合框架、Costless 和 RightFusion。为了确保实验结果的客观性，本文仅采集了未进行基础设施优化前的原始数据，以排除基础设施优化对函数融合效果的潜在影响。

图3.12展示了三个工作流进行函数融合后的成本累积分布函数（CDF）。与完全不进行函数融合的原生工作流架构相比，暴力搜索得到的融合最优解在成本上能为这三个工作流分别节约 45.5%、51.9% 和 61.0%。这一结果充分体现了函数融合在成本节约上的显著有效性。具体而言，函数融合通过减少远程调用次数、避免双重计费以及优化资源分配，显著降低了工作流的执行成本。

对于取消订单工作流，动态函数融合框架的平均开销为 7.28×10^5\$，Costless 的平均开销为 7.38×10^5\$，与暴力搜索最优解的平均开销 7.21×10^5\$ 相比较为接近。对于保留车票工作流，动态函数融合框架的平均开销为 6.34×10^5\$，Costless 的平均开销为 6.43×10^5\$，与暴力搜索最优解的平均开销 6.29×10^5\$ 相比较为接近。对于获取行程剩余车票工作流，动态函数融合框架的平均开销为 5.06×10^4\$，Costless 的平均开销为 5.12×10^4\$，与暴力搜索最优解的平均开销 5.17×10^4\$ 相比较为接近。细



(a) 取消订单工作流
(a) Cancel Order Workflow

(b) 保留车票工作流
(b) Reserve Ticket Workflow

(c) 获取行程剩余车票工作流
(c) Retrieve Remaining Tickets for Itinerary Workflow

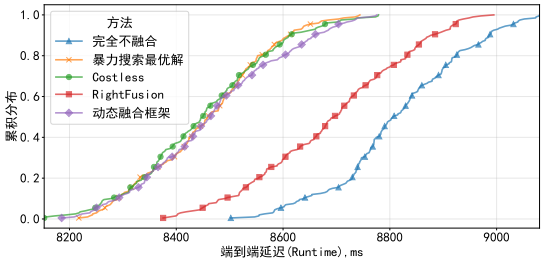
图 3.12　三个工作流进行函数融合后的成本 CDF
Figure 3.12　Cost CDF of the Three Workflows After Function Fusion

微的成本误差认为是实验机器波动所致，因为比较融合路径可以看出，对于这三个工作流，动态函数融合框架和 Costless 均找到了和暴力搜索最优解相同的融合解。

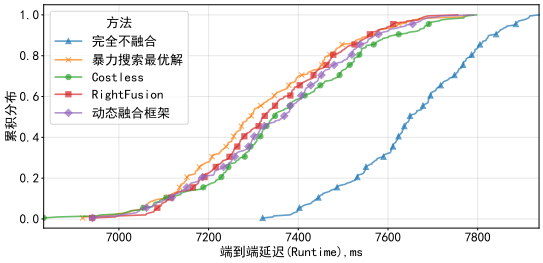
然而，RightFusion 在最优解的寻找上表现不佳，在取消订单工作流和获取行程剩余车票工作流中，其成本表现为 12.27×10^5\$ 和 11.23×10^4\$，仅在保留车票工作流中找到了最优解，得到了 6.32×10^5\$ 的平均开销。这表明 RightFusion 在处理嵌套工作流时存在一定的局限性，无法在复杂的工作流中取得较优解。

图3.13展示了三个工作流进行函数融合后的端到端时延累积分布函数 (CDF)。与完全不进行函数融合的原生工作流架构相比，暴力搜索得到的融合最优解在端到端时延上能为这三个工作流分别缩减 4.1%、4.3% 和 11.4%。这一结果充分体现了函数融合在缩短工作流端到端时延上的显著有效性。具体而言，函数融合通过减少远程调用次数、优化函数调用链以及减少级联冷启动开销，显著降低了工作流的端到端时延。

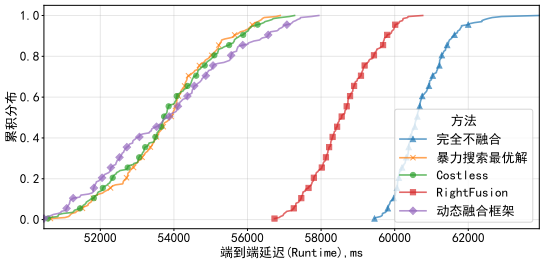
对于取消订单工作流，动态函数融合框架的平均端到端延迟为 8468.66 ms，Costless 的平均端到端延迟为 8451.53 ms，与暴力搜索最优解的平均端到端延迟 8457.26ms



(a) 取消订单工作流
(a) Cancel Order Workflow



(b) 保留车票工作流
(b) Reserve Ticket Workflow



(c) 获取行程剩余车票工作流
(c) Retrieve Remaining Tickets for Itinerary Workflow

图 3.13　三个工作流进行函数融合后的端到端时延 CDF
Figure 3.13　End-to-End Latency CDF of the Three Workflows After Function Fusion

相比较为接近。对于保留车票工作流，动态函数融合框架的平均端到端延迟为 7368.32 ms，Costless 的平均端到端延迟为 7356.72 ms，与暴力搜索最优解的平均端到端延迟 7340.09 ms 相比较为接近对于获取行程剩余车票工作流，动态函数融合框架的平均端到端延迟为 53812.39 ms，Costless 的平均端到端延迟为 53779.36 ms，与暴力搜索最优解的平均端到端延迟 53803.24 ms 相比较为接近细微的端到端误差同样认为是实验机器波动所致，因为比较融合路径可以看出，对于这三个工作流，动态函数融合框架和 Costless 均找到了和暴力搜索最优解相同的融合解。

然而，RightFusion 在最优解的寻找上表现不佳，在取消订单工作流和获取行程剩余车票工作流中，其端到端时延表现为 8740.83 ms 和 58592.45 ms，仅在保留车票工作流中找到了最优解，得到了 7335.11 ms 的平均端到端时延。这同样表明证明 RightFusion 在处理嵌套工作流时存在一定的局限性，所寻找的结果在端到端时延上也不占优势，不能寻找到较优解。

图3.14展示了三个工作流进行函数融合后的平均冷启动时间。与完全不进行函数融合的原生工作流架构相比，暴力搜索得到的融合最优解在冷启动时间上能为这三

个工作流分别缩减 76.2%、62.5% 和 90.1%。这一结果充分体现了函数融合在缩短级联冷启动时间上的显著有效性。具体而言，函数融合通过减少函数实例的创建次数、优化函数调用链以，显著降低了工作流的冷启动时间。

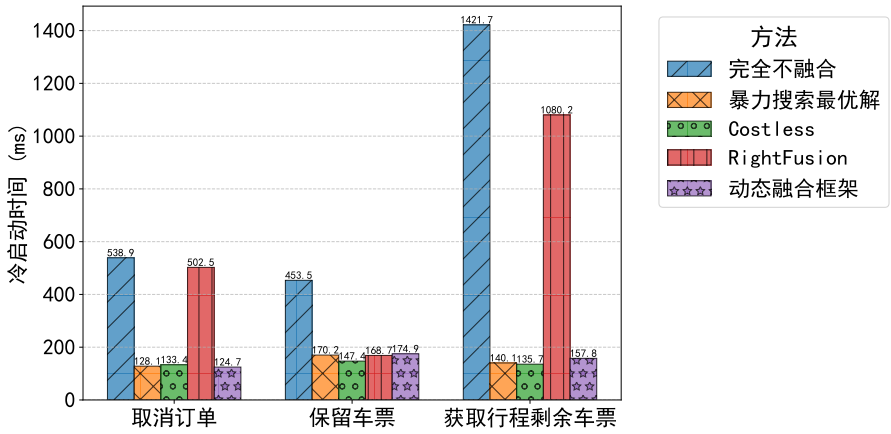


图 3.14　三个工作流进行函数融合后的平均冷启动时间
Figure 3.14　Average Cold Start Time of the Three Workflows After Function Fusion

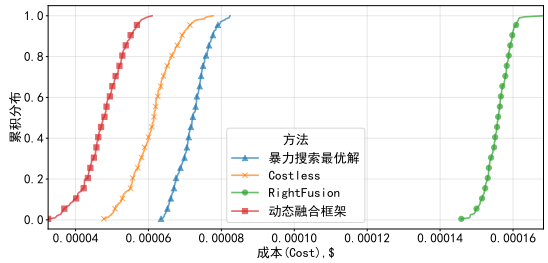
同时，比较动态函数融合框架与两个基线方法在冷启动时间上的表现，可以看出，动态函数融合框架与 Costless 均能找到暴力搜索的最优解，并获得类似的冷启动时间表现。同样的，由于 RightFusion 在最优解的寻找上表现不佳，在取消订单工作流和获取行程剩余车票工作流中，其冷启动时间表现与原生工作流架构相差无几，未能体现出明显的优化效果。

综上所述，以上融合实验的结果充分证明了在嵌套工作流中进行函数融合的有效性。函数融合不仅能够在减少双重计费、级联冷启动等方面取得显著优化，还能在成本和端到端时延上带来显著的性能提升。尤其是动态函数融合框架，能够在不进行暴力搜索的前提下，在多个不同类型的嵌套工作流中快速寻找到函数融合的最优解，为无服务器应用程序的性能优化提供了强有力的支持。与基线方法相比，动态函数融合框架在成本、端到端时延和冷启动时间等方面均表现出显著优势，进一步验证了其在实际应用中的有效性和普适性。

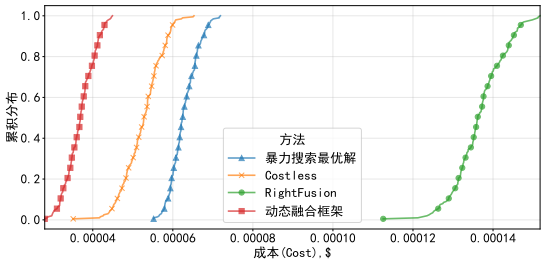
3.4.4.2　基础设施优化效果分析

为了进一步评估动态函数融合框架对于嵌套工作流的基础设施配置优化效果，对三个典型工作流（取消订单、保留车票和获取行程剩余车票）进行了详细的实验分析。在函数融合阶段结束后，动态函数融合框架、Costless 和 RightFusion 分别对工作

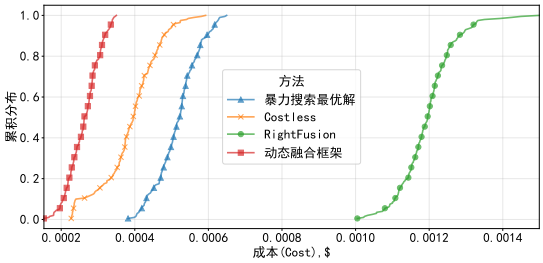
流中的各函数进行了基础设施的配置优化。



(a) 取消订单工作流
(a) Cancel Order Workflow



(b) 保留车票工作流
(b) Reserve Ticket Workflow

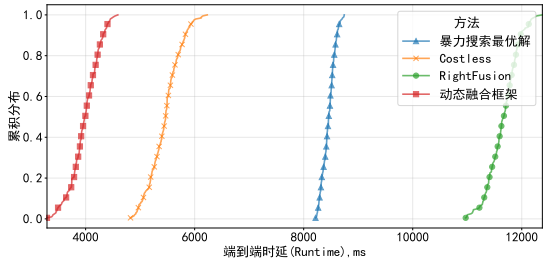


(c) 获取行程剩余车票工作流
(c) Retrieve Remaining Tickets for Itinerary Workflow

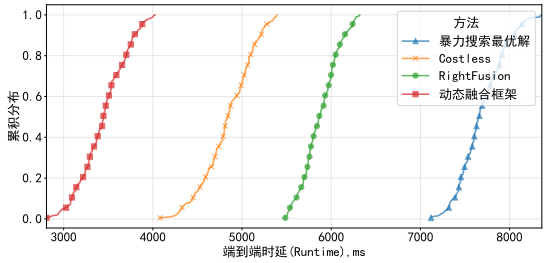
图 3.15　三个工作流进行基础设施优化后的成本 CDF
Figure 3.15　Cost CDF of the Three Workflows After Infrastructure Optimization

图3.15展示了对三个函数融合后的工作流进行基础设施优化后的成本累积分布函数（CDF）。三个工作流未进行基础设施优化的融合最优解的平均开销分别为 7.20×10^5\$、6.28×10^5\$、5.17×10^4\$；Costless 的平均开销分别为 6.61×10^5\$、5.25×10^5\$、3.88×10^4\$；RightFusion 的平均开销分别为 15.57×10^5\$、13.65×10^5\$、11.95×10^4\$；动态函数融合框架的平均开销分别为 4.76×10^5\$、3.68×10^5\$、2.62×10^4\$。

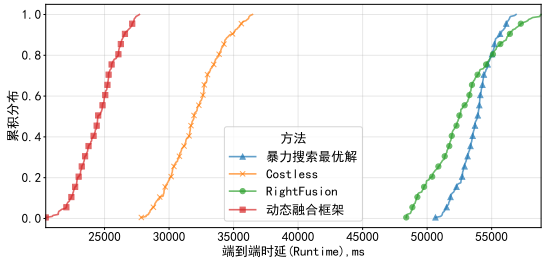
与未进行基础设施优化的融合最优解相比，动态函数融合框架在三个工作流的成本上平均还能分别优化 33.9%、41.4% 和 49.3%。这一结果表明，通过进一步优化基础设施配置，动态函数融合框架能够显著降低工作流的执行成本。具体而言，优化后的内存配置和 vCPU 分配减少了资源浪费，同时提高了函数的执行效率，从而实现了成本的进一步节约。相比之下，Costless 在基础设施优化方面的表现也能取得一定的效果，平均可以优化 8.2%、16.4% 和 24.9% 的成本。然而，RightFusion 由于在函数融合阶段无法寻找到最优解，因此在基础设施优化后也无法取得优于函数融合最优解的初始设置下的成本。



(a) 取消订单工作流
(a) Cancel Order Workflow



(b) 保留车票工作流
(b) Reserve Ticket Workflow



(c) 获取行程剩余车票工作流
(c) Retrieve Remaining Tickets for Itinerary Workflow

图 3.16　三个工作流进行基础设施优化后的端到端时延 CDF
Figure 3.16　End-to-End Latency CDF of the Three Workflows After Infrastructure Optimization

图3.16展示了对三个函数融合后的工作流进行基础设施优化后的端到端时延累积分布函数（CDF）。三个工作流未进行基础设施优化的融合最优解的平均端到端时延分别为 8457.26 ms 、7688.45 ms 、53803.24 ms ； Costless 的平均端到端时延分别为 5953.73 ms 、4839.54 ms 、32002.98 ms ； RightFusion 的平均端到端时延分别为 11656.93 ms 、5876.06 ms 、52724.11 ms ； 动态函数融合框架的平均端到端时延分别为 3983.96 ms 、3445.79 ms 、24489.32 ms ；

与未进行基础设施优化的融合最优解相比，动态函数融合框架在三个工作流的端到端时延上平均还能分别优化 52.9%、55.2% 和 54.5%。这一结果表明，通过进一步优化基础设施配置，动态函数融合框架能够显著缩短工作流的端到端时延。具体而言，优化后的内存配置和 vCPU 分配减少了函数执行的等待时间，从而实现了端到端时延的进一步缩减。Costless 在基础设施优化方面的表现与动态函数融合框架类似，平均也能优化 29.6%、37.1% 和 40.5% 的端到端时延。然而，RightFusion 由于在函数融合阶段无法寻找到最优解，因此在基础设施优化后也无法取得优于函数融合最优解的初始设置下的端到端时延。

综上所述，以上优化实验的结果充分证明了动态函数融合框架在嵌套工作流中进一步优化基础设施的有效性，表明仅依赖基础设施优化无法弥补函数融合阶段的不足。通过结合函数融合和基础设施优化，动态函数融合框架能够在成本和端到端时延上实现双重优化，显著提升工作流的性能和经济性。与基线方法相比，动态函数融合框架在基础设施优化方面表现出了显著的优势，进一步验证了其在实际应用中的有效性和普适性。

3.5　本章小结

本章首先深入分析了无服务器嵌套工作流部署中的关键问题，包括双重计费、级联冷启动、基础设施配置和远程调用开销等，并通过初步实验验证了函数融合的有效性。然后提出了一种反馈驱动的动态函数融合框架，通过结合函数融合和基础设施优化，实现了对嵌套工作流的全方位优化。实验结果进一步验证了该框架在成本和端到端时延优化方面的显著优势，尤其是在处理复杂嵌套工作流时，表现出了优于基线方法的性能。

第 4 章 基于资源解耦的工作流动态资源配置框架

随着无服务器计算的快速发展，FaaS 在构建复杂应用时面临资源管理效率低下的问题，以内存为中心的资源配置方式难以满足多样化工作负载的需求，而且单一的资源配置策略也无法充分提高输入敏感型工作流的成本效益。本章首先通过实验验证资源解耦的有效性，并分析输入特征对资源需求的影响。接着介绍基于资源解耦的工作流动态资源配置框架的设计与实现，包括核心模块的功能与协作机制。最后通过实验评估框架性能，并与基线方法进行对比分析，验证其在优化资源配置和降低成本方面的显著优势。

4.1 研究动机

本节初步验证无服务器工作流中资源解耦的有效性，并探讨输入特性对输入敏感型工作流资源需求的影响。

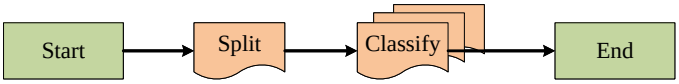
4.1.1 资源解耦有效性的验证

在无服务器计算环境中，资源管理是一个关键挑战。以内存为中心的资源配置（如 AWS Lambda）简化了资源管理，但它们可能并不适用于所有无服务器工作负载。Bilal 等人[47] 的研究表明，与耦合配置相比，资源解耦可以将执行成本降低高达 40%。然而，他们的分析仅限于单个函数，并未考虑工作流。为了填补这一研究空白，本节进行了实验探讨解耦 CPU-内存资源对工作流的运行时间和成本的影响。

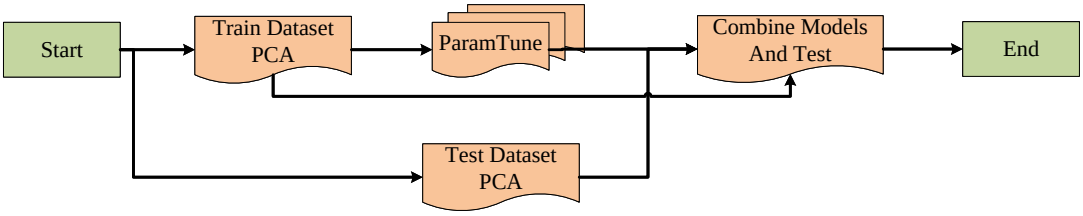
本章选择了三种不同的工作流[66]（Chatbot、ML Pipeline 和 Video Analysis）进行实验。图4.1展示了这些工作流的架构。

Chatbot 工作流主要用于处理用户输入并生成实时响应。该工作流首先接收用户输入，随后并行训练分类器以识别用户意图，并通过远程存储系统进行实时意图检测。Chatbot 工作流的设计体现了高并发和低延迟的需求，适用于需要快速响应的交互式应用场景。

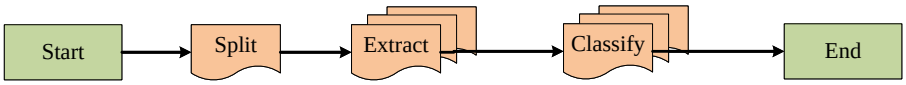
ML Pipeline 工作流专注于实现机器学习的完整流程。该工作流依次执行数据降维、模型训练和测试等关键步骤，涵盖了从数据预处理到模型评估的全过程。ML Pipeline 工作流展示了无服务器计算在数据密集型任务中的应用潜力，尤其适合需要高效资源分配的机器学习任务。



(a) Chatbot 工作流
(a) Chatbot Workflow



(b) ML Pipeline 工作流
(b) ML Pipeline Workflow



(c) Video Analysis 工作流
(c) Video Analysis Workflow

图 4.1　三类工作流的架构图
Figure 4.1　Architecture of the Three Types of Workflows

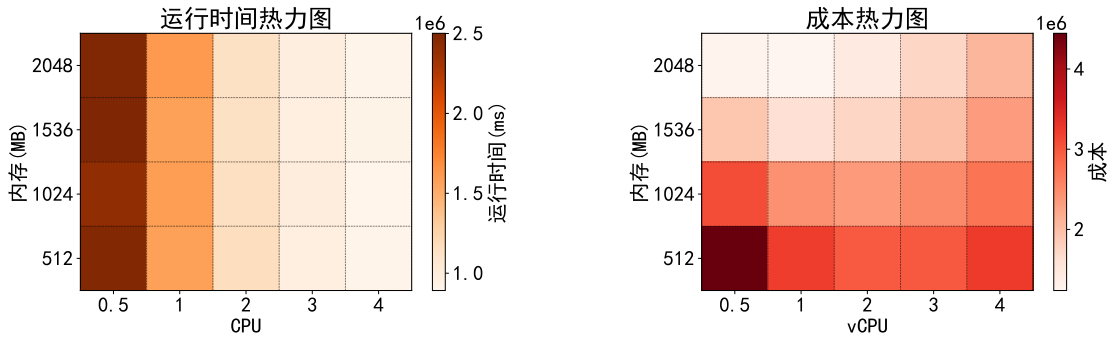
Video Analysis 工作流用于处理视频数据并提取关键信息。该工作流首先对输入视频进行分割，随后提取关键帧并对其进行分类分析。Video Analysis 工作流体现了无服务器计算在多媒体处理领域的优势，能够高效处理大规模视频数据，适用于视频内容分析和智能监控等场景。

对这三个工作流配置了不同的内存和 vCPU 进行测试，以验证内存、CPU 对工作流执行的影响，进而去观察不同工作流对资源的亲和度，以及其对成本的影响。

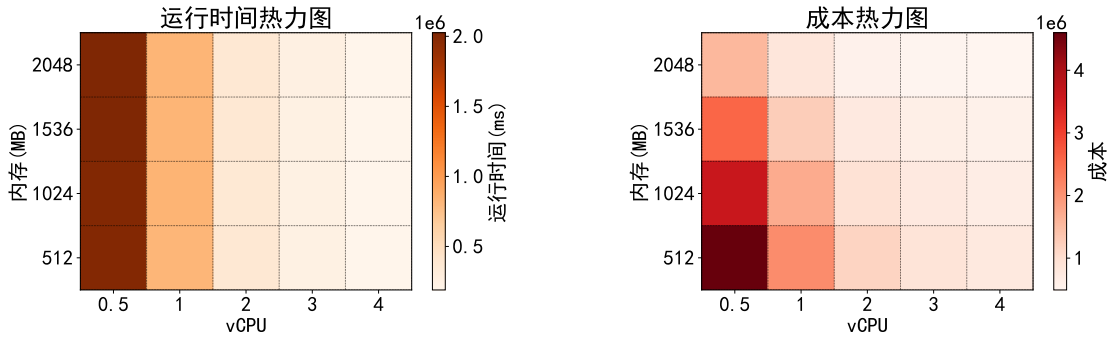
图4.2展示了 CPU 和内存解耦对工作流运行时间和成本的影响。发现尽管内存发生变化，Chatbot 和 ML Pipeline 的运行时间保持不变，如图4.2a和图4.2b所示。这表明以内存为中心的资源分配在计算密集型任务中效率低下。

在 Chatbot 工作流中，观察到内存的变化对运行时间几乎没有影响。这表明 Chatbot 工作流对内存的需求相对较低，而 CPU 的性能对其运行时间有显著影响。因此，解耦配置允许根据实际需求调整 CPU 资源，从而优化成本和性能。

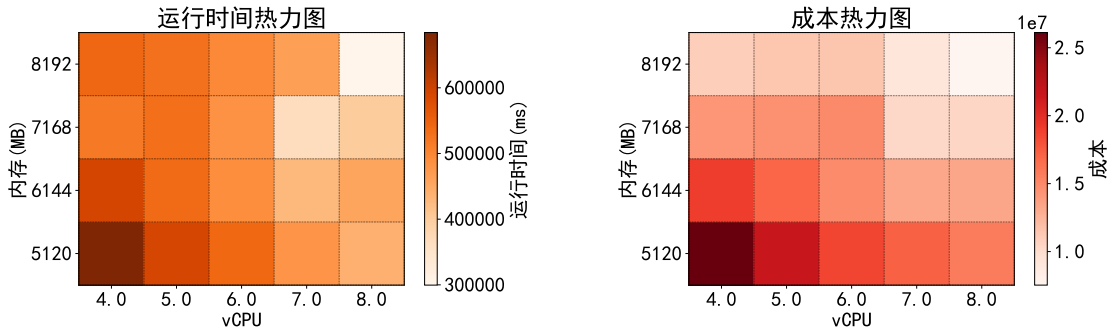
在 ML Pipeline 工作流中，发现 4 个 vCPU 和 512 MB 内存的解耦配置实现了最低成本，与耦合方法相比，内存使用量减少了 87.5%。这一结果突显了解耦策略的必



(a) Chatbot 工作流
(a) Chatbot Workflow



(b) ML Pipeline 工作流
(b) ML Pipeline Workflow



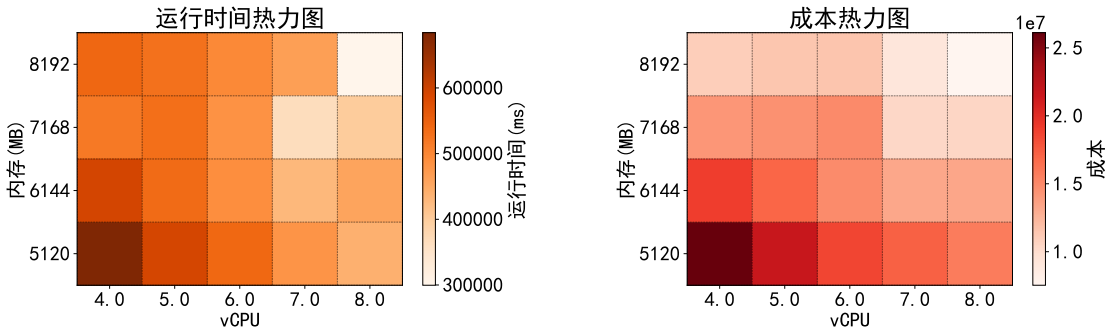
(c) Video Analysis 工作流
(c) Video Analysis Workflow

图 4.2 解耦资源策略下三类工作流的运行时长、成本热力图

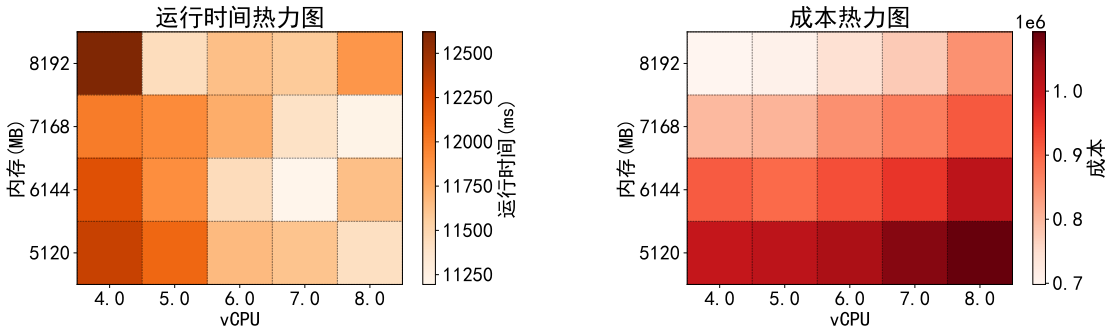
Figure 4.2 Heatmaps of Runtime and Cost for the Three Types of Workflows Under Decoupled Resource Strategy

要性，尤其是在计算密集型任务中。通过独立调整 CPU 和内存资源，能够显著降低成本，同时保持运行时间的稳定性。

此外，比较图4.2a和图4.2c可以看出，不同工作流对资源的亲和性存在显著差异。例如，Chatbot 以 512 MB 内存和 1 个 vCPU 实现成本最小化，而 Video Analysis 以



(a) 长视频输入的 Video Analysis 工作流
(a) Video Analysis Workflow for Long Video Inputs



(b) 短视频输入的 Video Analysis 工作流
(b) Video Analysis Workflow for Short Video Inputs

图 4.3　不同视频输入下的 Video Analysis 工作流的运行时长、成本热力图

Figure 4.3　Heatmaps of Runtime and Cost for the Video Analysis Workflow Under Different Video Inputs

5120 MB 内存和 8 个 vCPU 实现成本效率。这表明资源配置应根据每个工作流进行定制，以在满足 SLO 的同时最小化成本。

4.1.2　输入对函数资源需求影响的验证

在 Video Analysis 工作流中，输入视频的长度对资源需求和成本有着显著的影响。长视频输入通常需要更多的计算资源来处理，这不仅增加了运行时间，还显著提高了成本。相比之下，短视频输入所需的资源较少，运行时间和成本也相应降低。

图4.3展示了不同视频输入对 Video Analysis 工作流的影响。图4.3a显示了长视频输入下的运行时间和成本热力图，而图4.3b则展示了短视频输入下的情况。

对于长视频输入，观察到相同的 CPU 和内存需求下，所带来的运行时长飙升，由此带来了不一样的成本。

此外，不同视频输入对资源亲和度的差异也影响了最低成本点的分布。长视频

输入以 5120 MB 内存和 8 个 vCPU 资源配置下达到最低成本点，而短视频输入则在 8192 MB 内存和 4.0 个 vCPU 资源配置下达到最低成本点。这进一步强调了资源配置应根据输入特性进行动态调整的重要性，以在满足 SLO 的同时最小化成本。

4.2 基于资源解耦的工作流动态资源配置框架架构

本节将围绕基于资源解耦的工作流动态资源配置框架的设计展开讨论，阐述其设计目标、总体架构以及核心组件的工作流程。

4.2.1 设计目标

本章的总体设计目标是开发一种基于解耦内存和 CPU 的无服务器函数自动搜索方法，并针对输入敏感型的工作流能根据输入特性进行动态调整。

当前的无服务器函数资源搜索方法存在显著的局限性，主要体现在内存和 CPU 资源的捆绑分配上。这种捆绑分配方式使得用户无法根据应用程序的实际需求灵活分配所需的资源，从而导致了资源的浪费。此外，大多数开发者难以准确预测应用程序在不同资源配置下的性能和成本平衡点，这进一步加剧了资源管理的复杂性。为了解决这些问题，提出了一种基于解耦内存和 CPU 的资源自动配置方法。通过独立调整 CPU 和内存资源，能够根据应用程序的实际需求进行精细化配置，从而在满足 SLO 的同时，最大限度地降低成本。

现有的无服务器资源搜索方法通常忽略了输入特性对资源需求的影响，这导致了在处理不同输入时，资源的分配不够灵活和高效。例如，在 Video Analysis 工作流中，长视频输入和短视频输入对资源的需求存在显著差异，而现有的方法无法根据输入特性动态调整资源配置。为了应对这一挑战，提出了一种基于随机森林模型的动态资源配置方法。该方法通过分析输入特性与最佳资源配置之间的关系，能够根据不同的输入特性动态调整资源配置，从而在满足 SLO 的同时，实现成本的最小化。具体而言，首先通过历史数据训练随机森林模型，以捕捉输入特性与资源需求之间的复杂关系。然后，在实际运行过程中，系统会根据当前输入的特性，利用训练好的模型预测最佳的资源配置，并自动进行调整。这种方法不仅提高了资源利用效率，还显著降低了运行成本，特别是在处理输入敏感型工作流时，效果尤为显著。

4.2.2 总体架构

图4.4 展示了基于资源解耦的工作流动态资源配置框架 (Automated Affinity-aware Resource Configuration, AARC) 的总体架构，主要包括三个主要组件：工作流采样调

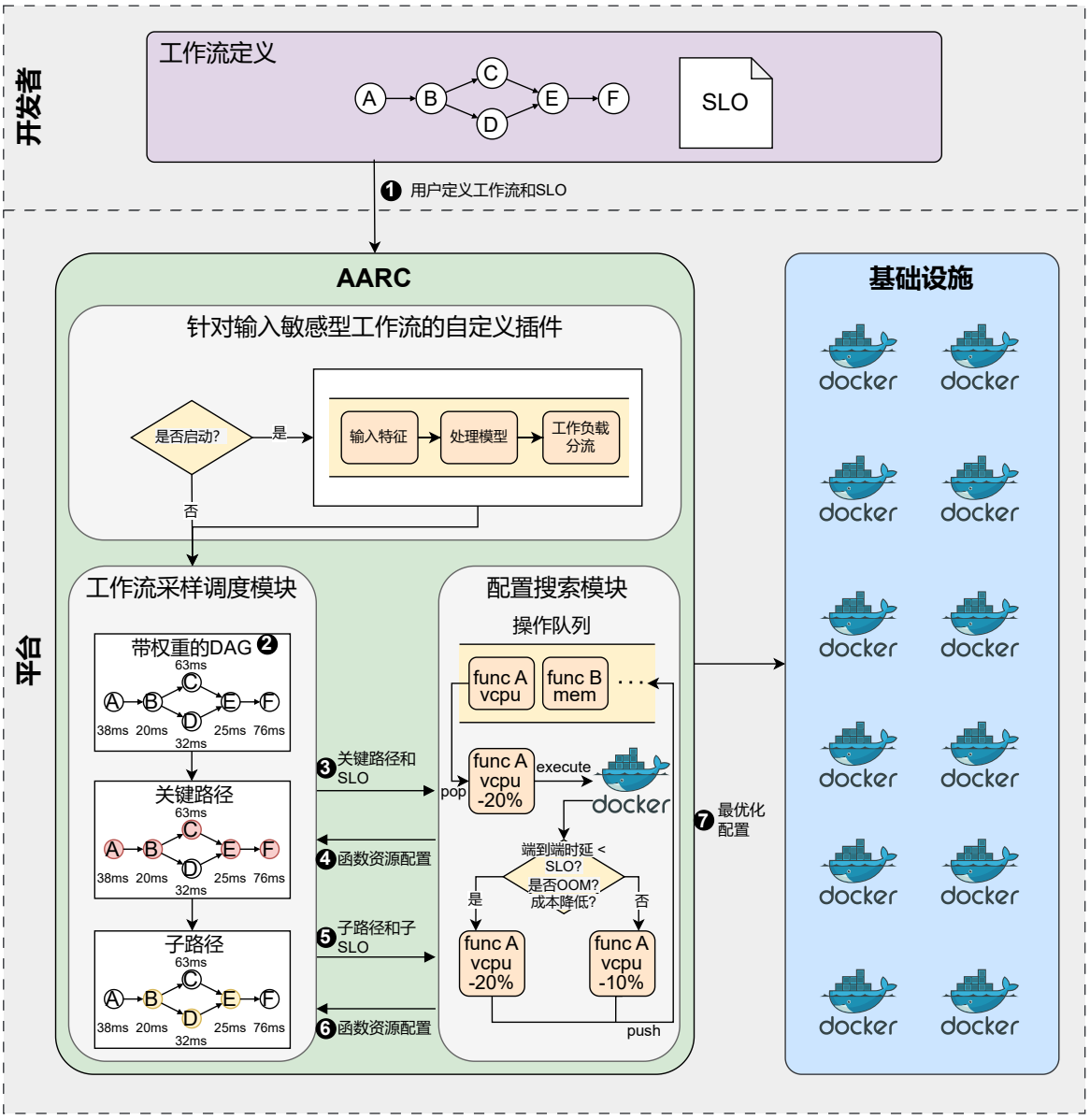


图 4.4　基于资源解耦的工作流动态资源配置框架总体架构

Figure 4.4　Overall Architecture of the Dynamic Resource Allocation Framework for Workflows Based on Resource Decoupling

度模块、基于优先级调度的配置解耦搜索模块和针对输入敏感型工作流的动态配置模块。

工作流采样调度模块负责分解输入的工作流，并识别其关键路径以及与之相关的所有子路径。配置搜索模块则负责配置关键路径和子路径上的每个函数。动态配置模块会识别工作负载的输入特征，提前根据输入特征将工作负载分流后进行配置搜

索，并依据配置搜索的结果建立起性能模型。

在 AARC 中，开发者首先将其工作流提交到云平台，并附带服务水平目标（SLO）。首先，工作流采样调度模块使用虚拟输入对用户定义的工作流进行分析，计算工作流中每个函数的运行时间。这些运行时间被用作每个节点的权重，将工作流转换为加权 DAG。接下来，调度模块从加权 DAG 中提取关键路径及其 SLO，并将这些信息传递给优先级配置器。配置搜索模块通过基于优先级的调度，逐步减少每个函数的内存和 CPU 分配，以确定满足 SLO 的最佳资源配置。基于关键路径和 SLO 的最佳配置，工作流采样调度模块将非关键路径上的函数替换到关键路径上，生成子路径及其相应的 SLO。同样，配置搜索模块计算子路径中函数的最佳资源配置，确保关键路径的 SLO 不被违反。最后，工作流采样调度模块最终确定资源配置，以便后续的容器资源分配，确保在最佳成本效率下满足 SLO。

为了进一步优化资源配置，特别是在处理输入敏感型工作流时，框架引入了基于随机森林模型的动态配置模块，该模块的主要功能是根据输入特性动态调整资源配置，从而在满足 SLO 的同时，实现成本的最小化。首先，系统收集工作负载的输入特性，输入特性差别较大的工作负载会进行分流，然后按照前文所述调用工作流采样调度模块和配置搜索模块。搜索完成后收集各输入特性所对应的函数资源配置，这些数据经过预处理后，用于训练随机森林模型，以捕捉输入特性与最佳资源配置之间的关系。随机森林模型能够处理复杂的非线性关系，从而更准确地预测最佳资源配置。在实际运行过程中，系统会根据当前输入的特性，利用训练好的随机森林模型预测最佳的资源配置。

4.3 基于资源解耦的工作流动态资源配置框架实现

本节将详细介绍基于资源解耦的工作流动态资源配置框架的实现细节，重点阐述其核心组件——工作流采样调度模块、基于优先级调度的配置解耦搜索模块以及针对输入敏感型工作流的动态配置模块的工作原理与设计思路。

4.3.1 工作流采样调度模块

工作流采样调度模块是整个资源自动配置框架的核心组成部分之一。该模块通过识别和优化工作流中的关键路径，确保在满足服务水平目标（SLO）的同时，最大限度地降低资源成本。关键路径的识别是整个调度过程的基础，因为它决定了工作流的瓶颈所在，从而为后续的资源优化提供了明确的方向。通过全 DAG 遍历和优先级

表 4.1　算法4.1 和算法4.2中使用到的函数
Table 4.1　Functions Utilized in Algorithm 4.1 and Algorithm 4.2

Functions	Description
deallocate(`op`***)***	根据给定操作 `op` 的类型和步骤，剥夺函数的一部分资源，并返回在此配置下的运行时间和成本。
allocate(`op`***)***	根据给定操作 `op` 的类型和步骤，为函数分配一部分资源，并返回新的步骤和操作轨迹。
find_critical_path(`G`***)***	给定一个加权 DAG，返回该 DAG 的关键路径。
find_detour_subpath(`G`, `critical_path`***)***	给定一个 DAG 及其关键路径，返回与关键路径相连的所有子路径。
runtime_sum(`path`, `start`, `end`***)***	给定路径的起点和终点，计算起点和终点之间的总持续时间。

调度，该模块不仅能够为关键路径上的函数生成优化的配置，还能够识别与关键路径相连的子路径，并对其进行合理的资源分配。这一过程不仅提高了资源利用效率，还确保了整个工作流的一致性和稳定性，从而在满足 SLO 的同时，实现了成本的最小化。

算法4.1 展示了整个调度过程。给定一个函数工作流和目标端到端 `SLO`，算法操作如下：

首先，工作流中的每个函数都被初始分配一个过度配置的基础配置，以确保工作流在大多数情况下满足 SLO（第 2-4 行）。这一步骤是至关重要的，因为初始配置的合理性直接影响到后续的优化过程。过度配置的基础配置虽然可能会导致一定的资源浪费，但它能够确保工作流在初始阶段就能够满足 SLO，从而为后续的优化提供一个稳定的基础。

然后，执行工作流，并将每个函数的运行时间用作 DAG 中的权重，以识别关键路径（第 5-6 行）。这一步骤通过实际执行工作流，获取每个函数的运行时间，并将其作为 DAG 中的权重，从而能够准确地识别出关键路径。关键路径的识别是整个调度过程的核心，因为它决定了工作流的瓶颈所在，从而为后续的资源优化提供了明确的方向。

关键路径和端到端 SLO 随后被用作配置搜索器的输入，后者依次为关键路径上的每个函数生成优化的配置（第 7-9 行）。配置搜索器在这一步骤中扮演了至关重要的角色，它通过优先级调度的方式，为关键路径上的每个函数生成优化的配置。这一

算法 4.1 基于关键路径算法的整体工作流调度

```
   Input: 工作流 G，端到端延迟 SLO
   Output: 每个函数的配置 G_configs
1  Func 调度 (G, SLO):
       // 分配基础配置
2      foreach v_i ∈ G do
3          v_i.config ← base_config
4      end
       // 执行以找到关键路径
5      执行 G
6      L ← find_critical_path(G)
7      G_configs ← {}
8      configs ← priority_configuration(L, SLO)
9      G_configs ← G_configs ∪ configs
       // 计算子路径的配置
10     subpaths ← find_detour_subpath(G, L)
11     foreach sp ∈ subpaths do
12         SLO′ ← runtime_sum(L, sp.start, sp.end)
13         foreach v_i ∈ sp do
14             if v_i 已调度 then
15                 v_i ← 弹出 (sp)
16                 SLO′ ← SLO′ − v_i.runtime
17             end
18         end
19         configs ← priority_configuration(sp, SLO′)
20         G_configs ← G_configs ∪ configs
21     end
22     return G_configs
23 结束
```

过程不仅考虑了函数的运行时间，还考虑了资源的成本，从而能够在满足 SLO 的同时，最大限度地降低成本。

通过全 DAG 遍历，算法识别与关键路径相连的子路径，由其在关键路径中的起点和终点定义，并且与其他节点没有交集（第 10 行）。这一步骤通过全 DAG 遍历，识别出与关键路径相连的所有子路径。子路径的识别是为了进一步优化资源配置，确保在关键路径之外的子路径也能够得到合理的资源分配，从而提高整体工作流的效率。

对于每个子路径，算法计算子 SLO 作为关键路径节点之间的时间间隔，确保调度期间关键路径的一致性（第 12 行）。这一步骤通过计算子路径的子 SLO，确保在调度过程中，关键路径的一致性得到维护。子 SLO 的计算是基于关键路径节点之间的

时间间隔，从而能够准确地反映出子路径的资源需求，为后续的资源配置提供依据。

配置搜索模块配置子路径上的每个函数，遵循与关键路径相同的方法（第 19-20 行）。这一步骤通过配置搜索模块对子路径上的每个函数进行配置，确保子路径上的函数也能够得到优化的资源配置。这一过程遵循与关键路径相同的方法，从而能够保持整个工作流的一致性，确保在满足 SLO 的同时，最大限度地降低成本。

为了防止冲突并确保每个函数只调度一次，算法为每个函数设置一个已调度标志。调度函数后，算法将其从路径中移除并相应调整 SLO（第 13-18 行）。这一步骤通过设置已调度标志，防止函数被重复调度，从而避免资源冲突。调度函数后，算法将其从路径中移除，并相应调整 SLO，确保在调度过程中，资源的分配和 SLO 的维护得到有效管理。

4.3.2　基于优先级调度的配置解耦搜索模块

基于优先级调度的配置搜索模块是整个资源自动配置框架的另一个核心组成部分。该模块通过优先级调度算法，实现了受服务水平目标（SLO）限制的顺序执行函数的资源优化。优先级调度的引入，使得算法能够在有限的资源下，快速找到最优的配置，从而在满足 SLO 的同时，最大限度地降低成本。通过初始化优先队列、动态调整资源、恢复资源、指数退避机制和迭代限制等步骤，该模块不仅提高了资源利用效率，还确保了整个工作流的一致性和稳定性。

算法4.2 展示了所设计的配置搜索算法，该算法采用优先级调度来优化受 SLO 限制的顺序执行函数的配置。

算法初始化一个优先队列 `PQ` 来管理每个函数的操作（第 2 行），这需要两个不同的操作来调整 CPU 和内存配额（第 3-10 行）。优先队列的引入是为了高效地管理函数的资源调整操作，确保在有限的资源下，能够快速找到最优的配置。每个函数的操作包括调整 CPU 和内存配额，这些操作被初始化为具有最高优先级，以便在后续的调度过程中能够优先处理。

根据操作类型，算法修改函数的资源，不断从 `PQ` 执行操作并监控其对运行时间和成本的影响（第 12-13 行）。这一步骤通过不断从优先队列中取出操作，并根据操作类型调整函数的资源，从而实现资源的动态分配。每次操作执行后，算法会监控其对运行时间和成本的影响，以确保在满足 SLO 的同时，最大限度地降低成本。

如果操作违反 SLO、增加成本或遇到错误，算法将恢复资源（第 14-18 行）。这一步骤通过恢复资源，确保在操作失败或违反 SLO 的情况下，能够及时回滚，避免资源的浪费和系统的不稳定性。恢复资源的过程不仅包括撤销操作，还包括调整操作

算法 4.2 基于优先级调度的配置搜索算法

```
   Input: 函数路径 L，端到端延迟 SLO
   Output: 每个函数的配置 configs
1  Function priority_configuration(L, SLO):
2      初始化 PQ, count
3      foreach v_i ∈ L do
4          for type ∈ [cpu, mem] do
5              func, priority ← v_i, ∞
6              step, trial ← 1, FUNC_TRIAL
7              op ← {func, type, step, trial}
8              push (PQ, op, priority)
9          end
10     end
11     while len(PQ) > 0 AND count < MAX_TRAIL do
12         op, count ← pop(PQ), count + 1
13         runtime, cost ← deallocate(op)
14         if runtime > SLO or cost 增加 then
15             op.trial, op.step ← allocate(op)
16             if op.trial > 0 then
17                 push(PQ, op, 0)
18             end
19         else
20             priority ← 减少的 cost
21             push(PQ, op, priority)
22         end
23     end
24     configs ← {(v_i.cpu, v_i.mem) | v_i ∈ L}
25     return configs
26 结束
```

的优先级和步长，以便在后续的调度过程中能够更加谨慎地处理。

同时，指数退避机制减少了步长以确保收敛，同时避免过度调度（第 15 行）。指数退避机制的引入是为了在资源调整过程中，逐步减少步长，从而确保算法的收敛性。通过逐步减少步长，算法能够在有限的迭代次数内，找到最优的资源配置，同时避免过度调度导致的资源浪费。

持续导致违规的操作（`trail = 0`）将从 `PQ` 中移除（第 16 行），而那些有潜力的操作将重新入队并调整优先级（第 17、20-21 行）。这一步骤通过移除持续导致违规的操作，确保优先队列中的操作都是具有潜力的，从而提高调度效率。对于那些有潜力的操作，算法会重新入队并调整优先级，以便在后续的调度过程中能够优先处理。

循环在 PQ 为空或达到用户定义的迭代限制 $\mathrm{MAX}_{\mathrm{TRAIL}}$ 时终止（第 11 行）。这一步骤通过设置迭代限制，确保算法在有限的迭代次数内完成调度过程，从而避免无限循环导致的资源浪费。当优先队列为空或达到迭代限制时，算法将终止，并输出最终的资源配置。

4.3.3　针对输入敏感型工作流的动态配置模块

针对输入敏感型工作流的动态配置模块是资源自动配置框架中的核心组件之一，旨在通过动态调整资源配置来优化系统性能并降低成本。

表 4.2　算法 4.3 中的函数
Table 4.2　The Functions in Algorithm 4.3

Functions	Description
identify_features(F***)***	根据输入特征 F，提取所有可能的特征。
feature_selection(features***)***	使用特征选择算法（如信息增益或主成分分析）筛选关键特征。
split_workloads(selected_features***)***	根据输入特征的差异性将工作负载分流为多个组。
sample_scheduling(group, SLO***)***	为分流后的工作负载组调用工作流采样调度模块，获取解耦后的最佳配置。
custom_model(features, D***)***	开发者自定义的模型，根据特征 features 和历史数据 D 构建。
train_random_forest(configurations, D***)***	使用随机森林模型，根据配置搜索情况 configurations 和历史数据 D 训练模型。
predict_resources(model, features***)***	基于模型 model 和输入特征 features，预测工作流各函数的最优资源配置。

该模块的核心思想是通过对输入数据的特征进行深度分析，将工作负载根据其特征的差异性分流为多个组，从而实现对不同工作负载的精细化管理和资源配置优化。具体而言，模块首先从输入数据中提取多维特征（如视频的分辨率、帧率、编码格式，或数据集的规模、复杂度、分布特性等），并利用特征选择算法（如基于信息增益、主成分分析或递归特征消除的方法）筛选出对资源配置影响最为显著的关键特征。这一步骤不仅减少了后续模型训练的复杂度，还确保了资源配置策略能够准确地反映输入数据的特性。

在完成特征提取与选择后，模块根据输入特征的差异性，将工作负载分流为多个组。例如，高分辨率视频和低分辨率视频可能被分为不同的组，或者大规模数据集和小规模数据集可能被分配到不同的处理路径。这种分流策略不仅能够减少配置搜索的复杂度，还能够为每个组单独优化资源配置，从而显著提高资源分配的效率。分流

后，模块为每个工作负载组调用工作流采样调度模块，通过实验或模拟的方式获取解耦后的最佳配置。采样调度模块通过多轮迭代和优化，为每个组找到满足服务级别目标（SLO）的资源配置方案，从而确保系统在高负载和低负载情况下均能保持高效运行。

算法 4.3 输入敏感型动态配置插件

```
Input: 输入特征 F, 历史数据 D, 服务级别目标 SLO
Output: 优化后的资源配置 C_optimized
Function input_aware_configuration(F, D)SLO:
    // 步骤 1: 输入特征提取与选择
    features ← identify_features(F)
    selected_features ← feature_selection(features)
    // 步骤 2: 根据输入特征的差异性，提前为工作负载分流
    workload_groups ← split_workloads(selected_features)
    // 步骤 3: 为分流后的工作负载调用工作流采样调度模块，获得解耦后的最佳配置
    configurations ← ∅
    foreach group ∈ workload_groups do
        config ← sample_scheduling(group, SLO)
        configurations ← configurations ∪ config
    end
    // 步骤 4: 如果开发者未自定义模型，使用配置搜索情况训练随机森林模型以捕捉非线性关系，否则使用开发者的模型
    if custom_model 已定义 then
        model ← custom_model(selected_features,D)
    else
        model ← train_random_forest(configurations,D)
    end
    // 步骤 5: 根据输入特征预测工作流各函数资源配置
    C_optimized ← predict_resources(model,selected_features)
    return C_optimized
结束
```

随后，模块利用机器学习模型（如随机森林、梯度提升树或深度神经网络）捕捉输入特征与资源配置之间的非线性关系。随机森林模型因其鲁棒性和对高维数据的处理能力而被作为默认选择，能够有效处理复杂的非线性关系和多维特征之间的交互作用。如果开发者对输入特征与资源配置之间的关系有更深入的理解，模块还支持使用自定义模型（如基于贝叶斯优化或强化学习的模型）来替代默认模型。这种灵活性使得模块能够适应不同场景和需求，从而提供更加精准的资源配置预测。

模块基于训练好的模型和输入特征，预测工作流各函数的最优资源配置。预测结

果包括 CPU、内存等资源的分配方案，还能够根据实时负载情况进行动态调整。通过这种方式，模块能够在满足服务级别目标（SLO）的同时，最大限度地降低成本，并提高资源利用效率。

算法 4.3 展示了输入敏感型动态配置插件的工作原理。给定输入特征 `F`、历史数据 `D` 和服务级别目标 `SLO`，算法操作如下：

首先，算法调用 ***identify_features()*** 数，根据输入特征 `F`，提取所有可能的特征（第 2 行）。随后，算法调用 ***feature_selection()*** 数，使用特征选择算法（如信息增益或主成分分析）筛选出对资源配置影响最大的关键特征（第 3 行）。这一步骤确保了算法能够准确地反映输入对资源需求的影响，并为后续的分流和配置优化提供基础。

接下来，算法调用 ***split_workloads()*** 数，根据输入特征的差异性，将工作负载分流为多个组（第 5 行）。例如，高分辨率视频和低分辨率视频可能被分为不同的组。这一步骤的目的是减少配置搜索的复杂度，并为每个组单独优化资源配置。分流后，算法为每个工作负载组调用 ***sample_scheduling()*** 数，获取解耦后的最佳配置（第 7-9 行）。采样调度模块通过实验或模拟，为每个组找到满足服务级别目标（SLO）的资源配置方案。

在获取各组的最佳配置后，算法根据配置搜索情况训练机器学习模型。如果开发者定义了自定义模型，算法将调用 ***custom_model()*** 数，根据特征 `selected_features` 和历史数据 `D`，构建模型（第 12-13 行）。否则，算法将使用默认的随机森林模型，调用 ***train_random_forest()*** 数，训练模型（第 15 行）。随机森林模型能够处理高维数据和非线性关系，适合用于复杂的资源配置预测任务。

最后，算法调用 ***predict_resources()*** 数，基于训练好的模型 `model` 和输入特征 `selected_features`，预测工作流各函数的最优资源配置（第 18 行）。预测结果将用于动态调整资源分配，从而在满足 SLO 的同时，最大限度地降低成本。这一步骤确保了算法能够实现高效的资源分配，同时支持开发者自定义模型以满足特定需求。

4.4　实验分析

本节将通过实验验证基于解耦资源配置的动态工作流优化框架的有效性。首先介绍了实验的软硬件环境，接着描述了两种基线方法——贝叶斯优化（Bayesian Optimization, BO）和 MAFF 梯度下降算法，作为对比基准，以评估提出的框架在性能和成本优化方面的优势。随后，设计了基于三种典型工作流的实验，分别设置了不同的 SLO ，评估了解耦资源配置对工作流运行时间和成本的影响。最后，通过对实验结

果的分析，验证了 AARC 在满足 SLO 的同时，显著降低运行时间和成本的潜力，为无服务器工作流的资源配置优化提供了有力的实践支持。

4.4.1 实验软硬件环境

实验在一台配备 4 个 Intel Xeon Gold 6248R 处理器（96 个物理核心）和 512GB 内存的机器上运行。工作流在独立的 Docker 容器中执行，实现了 CPU 和内存分配的解耦。这种环境设置使得能够在不同的资源配置下运行工作流，从而评估不同配置对性能和成本的影响。

4.4.2 基线方法介绍

将 AARC 与两种基线方法进行比较：BO 和 MAFF 梯度下降算法。这两种方法最初设计用于单个函数的资源配置和以内存为中心的任务，但在此实验中，对其进行了适应性调整，以用于工作流优化。

在贝叶斯优化方法中，配置参数被离散化以限制搜索空间。内存分配以 64 MB 为增量，从 128 MB 到 10,240 MB，而 vCPU 核心数范围从 0.1 到 10，独立于内存。

MAFF 梯度下降方法通过迭代最小化成本，按比例分配 vCPU 核心（每 1,024 MB 内存分配 1 个核心）。如果工作流的 SLO 被违反，过程将回滚到上一步并终止。

4.4.3 搜索及测试实验介绍

实验使用了三种工作流（Chatbot、ML Pipeline 和 Video Analysis），分别设置了 120 秒、120 秒和 600 秒的 SLO。

使用运行时间和成本指标来评估性能，扩展了 AWS Lambda 的定价模型以适应解耦资源。设 $\mathtt{cost}_{ij}$ 表示配置为 $(\mathtt{cpu}_j, \mathtt{mem}_j)$ 的服务器函数 $\mathtt{v}_i$ 的运行时间 $\mathtt{t}_{ij}$ 的成本。用 μ_0 表示每 vCPU-秒的价格，μ_1 表示每 GB-秒的价格，μ_2 表示函数请求和编排的价格。所有这些都是常数。成本方程为

$$\mathtt{cost}_{ij} = \mathtt{t}_{ij}(\mu_0 \cdot \mathtt{cpu}_j + \mu_1 \cdot \mathtt{mem}_j) + \mu_2 \tag{4-1}$$

在这里，μ_0、μ_1 和 μ_2 分别设置为 0.512、0.001 和 0。

在配置搜索实验中，使用采样的运行时间和成本作为指标。在测试具有最佳配置的实验中，使用端到端延迟和成本作为指标。通过这些实验，能够评估不同配置对工作流性能和成本的影响，从而验证 AARC 在满足 SLO 的同时，最大限度地降低成本的有效性。

4.4.4　测试结果及分析

4.4.4.1　配置搜索模块有效性验证

在配置搜索实验中，为每个工作流设置了初始配置。然后，使用三种方法进行配置搜索：AARC、BO 和 MAFF 梯度下降。

为了验证 AARC 在满足 SLO 要求和执行成本方面的性能，使用上述方法生成的配置执行工作流 100 次，并计算其平均运行时间和成本。

表 4.3　工作流平均运行时间和成本
Table 4.3　Average Runtime and Cost of Workflows

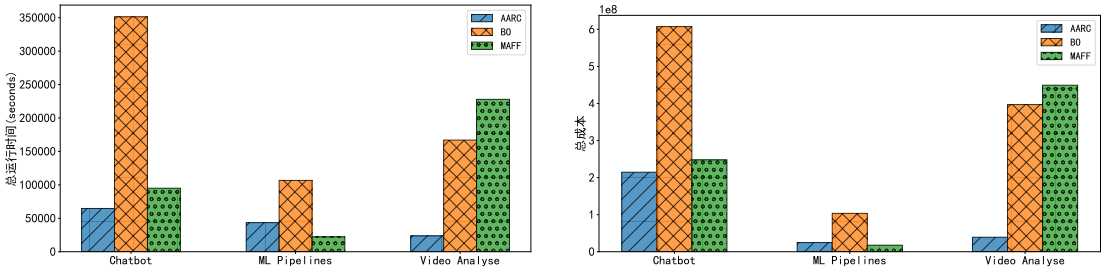
	Chatbot		**ML Pipeline**		**Video Analysis**	
	端到端时延 (s)	成本	端到端时延 (s)	成本	端到端时延 (s)	成本
AARC	103.7±3.2	2390.9k	77.1±2.6	435.0k	316.8±6.6	53.6k
BO	114.7±1.9	4275.2k	60.0±0.7	863.5k	519.9±8.3	82.4k
MAFF	115.3±3.1	3477.5k	109.5±2.0	1136.6k	578.2±19.3	98.8k

所有方法都满足 SLO 约束，平均运行时间和标准差如表 4.3 所示。AARC 满足 SLO 是因为算法在配置搜索过程中，当 SLO 违反时恢复资源，并将子路径纳入关键路径，确保了关键路径的一致性和 SLO 的合规性。这证明了 AARC 在满足 SLO 要求方面的有效性，并具有集成到具有高可靠性和性能标准的实际系统中的潜力。

表4.3 还显示了使用不同方法找到的配置执行工作流的成本。在 Chatbot、ML Pipeline 和 Video Analysis 工作流中，AARC 的成本显著降低，分别比贝叶斯优化和 MAFF 降低了 44.0% 和 31.2%、49.6% 和 61.7%、34.9% 和 45.7%。与 BO 和 MAFF 的比例配置方法相比，AARC 使用优先级调度允许更稳定地识别合适的解耦资源配置，实现了解耦资源分配从而获得了更低成本的配置。由于 ML Pipeline 对 CPU 需求高而对内存需求低，解耦资源分配产生了更好的结果。

4.4.4.2　配置搜索模块性能比较

图4.5 展示了采样过程的总运行时间和成本。在所有工作流中，AARC 优于贝叶斯优化，特别是在 Video Analysis 工作流中，运行时间减少了 85.8%，成本减少了 90.1%。这种改进源于 AARC 的优先级调度策略，该策略扩展了配置搜索空间，同时需要更少的样本。在 Chatbot 工作流中，AARC 执行了 64 次采样，略多于 MAFF 的 61 次，但运行时间减少了 31.9%，成本减少了 13.4%。这是因为 MAFF 的比例分配方案减少了搜索空间，但存在局部最优的风险，导致由于资源耦合而运行时间和成本更高。在 Video Analysis 工作流中，AARC 将运行时间和成本分别减少了 89.6% 和 91.3%，优



(a) 总运行时间对比图　　(b) 总成本对比图
(a) Total Runtime Comparison Chart　　(b) Total Cost Comparison Chart

图 4.5　采样过程总运行时间和成本
Figure 4.5　Total Runtime and Cost of the Sampling Process

于 MAFF。然而，在 ML Pipeline 工作流中，AARC 采样了 50 次，而 MAFF 采样了 15 次，导致 MAFF 的运行时间和成本更低。这是由于 ML Pipeline 对 CPU 需求高而对内存需求低，比例调整往往陷入局部最优，因此需要更少的采样来满足退出条件。

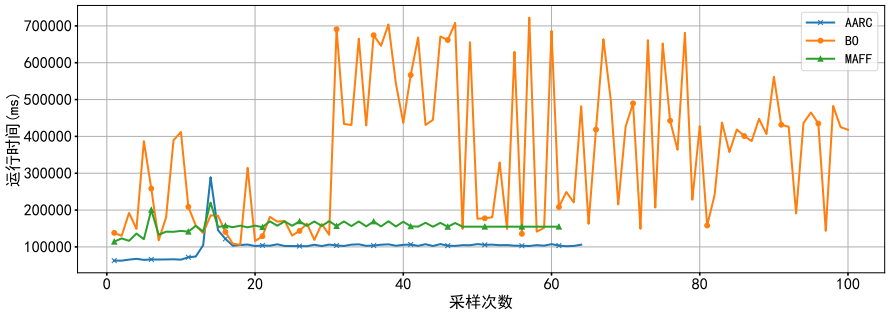
进一步展示了随着采样次数增加，每种方法计算出的最佳配置的有效性，通过工作流的运行时间和成本评估这些结果。图4.6 显示了不同配置下每个工作流的运行时间随采样次数的变化。设置目标是满足 SLO 的同时最小化成本，因此使用 AARC 时，运行时间呈上升趋势。贝叶斯优化方法的采样次数最多且不稳定，其效率下降是由于资源解耦创建了较大的搜索空间。相比之下，AARC 的优先级调度策略保持了搜索效率，同时加速了收敛。

图4.7 描绘了不同配置下成本随采样次数的变化。使用 AARC 时，成本呈下降趋势，并在更少的采样次数下收敛。在 ML Pipeline 工作流中，MAFF 方法由于其耦合资源配置搜索方法，很快陷入局部最优，难以发现更具成本效益的配置。AARC 通过采用解耦资源配置搜索，识别出更低成本的配置。

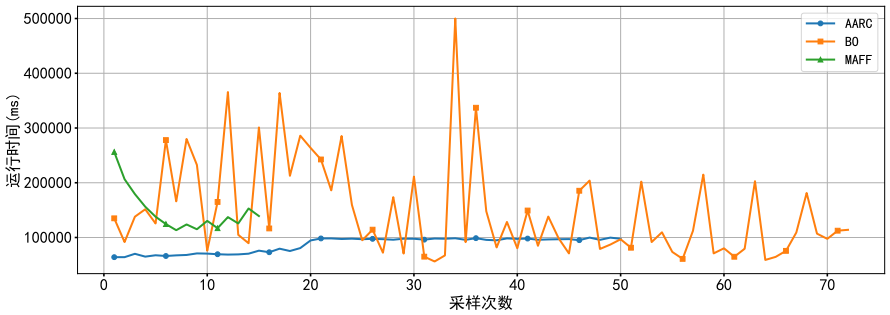
4.4.4.3　动态配置模块有效性验证

为了验证动态配置模块的有效性，本节采用三种不同长度的视频输入（20 秒、1 分钟和 6 分钟）作为 Video Analysis 工作流的典型工作负载，并启用动态配置模块寻找最佳配置。配置搜索结束后，使用每种长度的视频各 100 个作为模拟的用户请求，然后收集统计请求的端到端时延和开销。

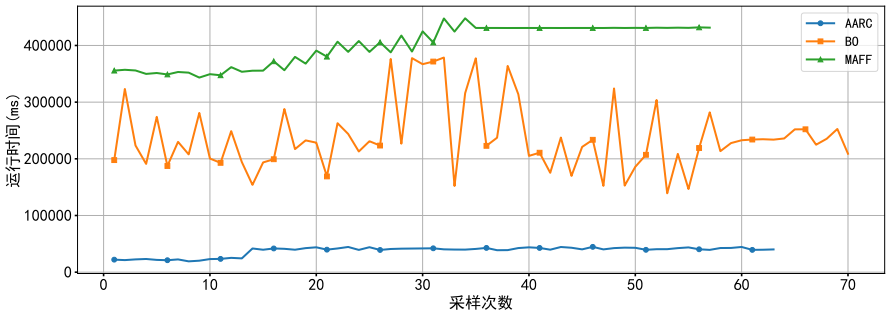
图4.8 展示了不同方法下不同请求的 Video Analysis 工作流的运行时间。在短视频和中视频输入下，三个方法的运行时间没有明显差别，而在长视频输入下则体现出了较大差别，只有 AARC 取得了较好的性能，与所设置的 SLO 要求相差较远。而两个基线方法则表现较差，MAFF 方法下少数长视频请求还发生了 SLO 超时。



(a) Chatbot 工作流
(a) Chatbot Workflow



(b) ML Pipeline 工作流
(b) ML Pipeline Workflow

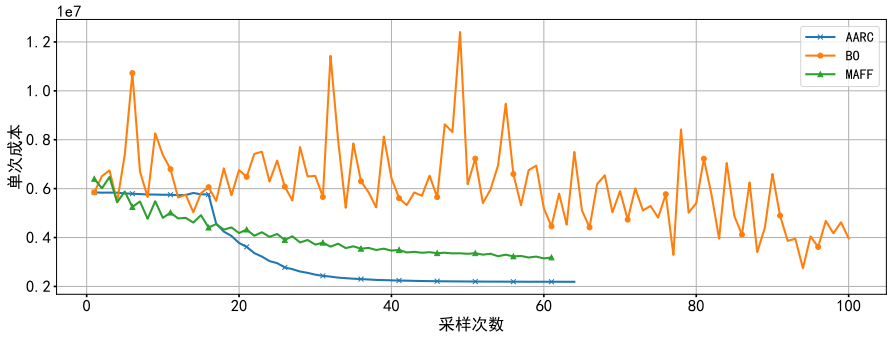


(c) Video Analysis 工作流
(c) Video Analysis Workflow

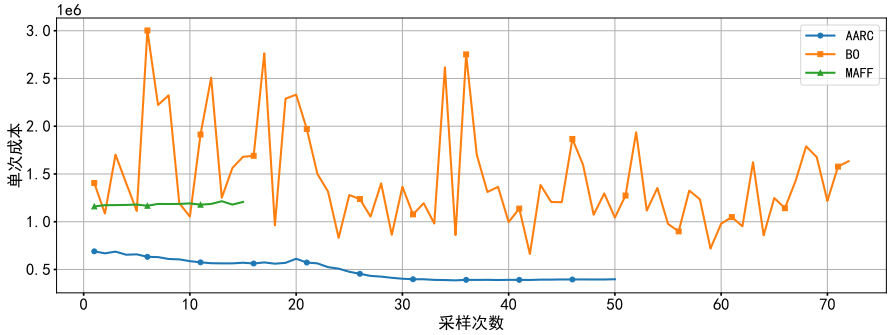
图 4.6　运行时间随着采样次数的变化
Figure 4.6　Runtime Variation with the Number of Sampling Iterations

表 4.4　不同方法下 Video Analysis 工作流的平均开销
Table 4.4　Average Cost and Cost of Workflows

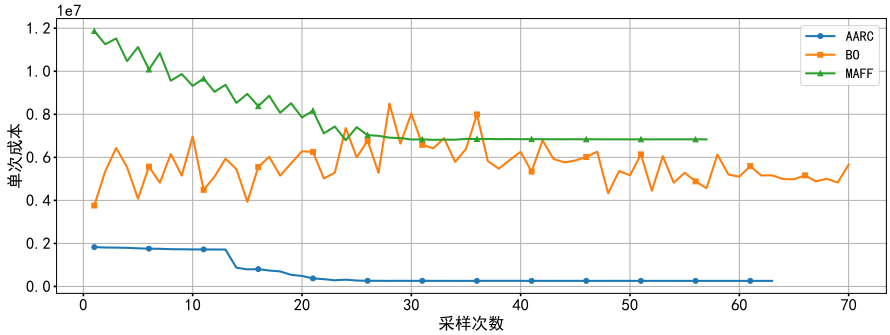
	短视频请求	中视频请求	长视频请求
AARC	71.2k	271.8k	5365.6k
BO	698.0k	1729.4k	8246.4k
MAFF	703.0k	2235.7k	9885.0k



(a) Chatbot 工作流
(a) Chatbot Workflow



(b) ML Pipeline 工作流
(b) ML Pipeline Workflow



(c) Video Analysis 工作流
(c) Video Analysis Workflow

图 4.7　成本随着采样次数的变化
Figure 4.7　Cost Variation with the Number of Sampling Iterations

表 4.4 展示了不同方法（AARC、BO 和 MAFF）在处理短视频、中视频和长视频请求时，Video Analysis 工作流的平均开销。从表中可以看出，AARC 方法在所有类型的视频请求中均显著降低了成本。其中，短视频请求的成本比起 BO 降低了 89.8%，比起 MAFF 降低了 89.9%；中视频请求的成本比起 BO 降低了 84.3%，比起 MAFF 降

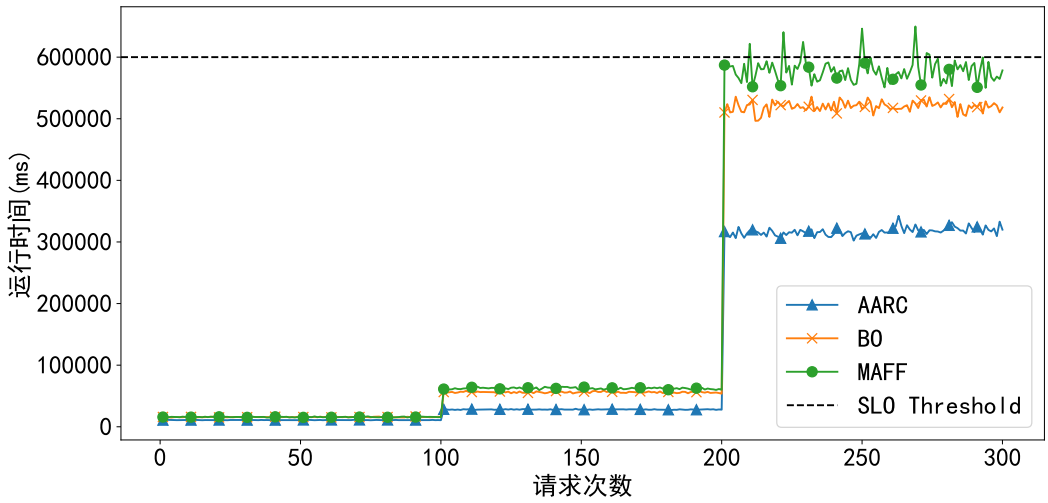


图 4.8　不同方法下 Video Analysis 工作流的运行时间
Figure 4.8　Runtime of Video Analysis Workflow Under Different Methods

低了 87.8%；长视频请求的成本比起 BO 降低了 34.9%，比起 MAFF 降低了 45.7%。综合来看，AARC 在 Video Analysis 工作流上的平均成本比起 BO 降低 45.6%，比起 MAFF 降低了 55.5%。

AARC 在输入敏感型工作流中的成本优势尤为突出，这主要归因于其随机森林模型对输入特征与资源配置之间非线性关系的高效捕捉。通过训练随机森林模型，AARC 能够准确预测不同长度下视频处理所需的资源配置，从而在满足 SLO 的同时，采用所搜索出来的适用于该长度视频的最佳配置，避免了资源的过度分配或分配不足，显著提高资源分配的效率，最大限度地降低成本。

4.5　本章小结

本章首先通过实验验证资源解耦的有效性，并分析输入特征对资源需求的影响。然后提出一种基于资源解耦的工作流动态资源配置框架，通过解耦内存和 CPU 资源，并结合输入敏感型工作流的特性，根据输入特性动态调整资源配置以优化性能和成本。实验验证表明，该框架在满足 SLO 的同时，显著降低了运行时间和成本，尤其在输入敏感型工作流中表现出更高的资源利用效率。与贝叶斯优化和 MAFF 梯度下降方法相比，所提出的框架在多种工作流中均展现出更优的性能和成本效益，为无服务器工作流的资源管理提供了有效的解决方案。

第 5 章 全文总结与展望

5.1 全文总结

本文针对无服务器工作流中的部署低效执行问题、资源捆绑分配问题以及输入敏感型工作流的动态资源配置问题展开研究，旨在提升工作流的性能和成本效益。首先，开发者在开发无服务器应用程序时通常将复杂任务分解为细粒度的、无状态的函数，然而这种细粒度分解与实例之间的一一映射不是最优解，尤其是在嵌套工作流中，可能导致双重计费、级联冷启动等问题，显著影响工作流的性能和成本效益。其次，大部分第三方供应商只为开发者提供捆绑的资源配置选项，这种策略不仅给开发者带来了资源配置负担，而且工作流的资源亲和性差异会显著影响在自动配置过程中所能达到的性能和成本效益。此外，输入敏感型工作流中，不同输入特征（如视频长度、数据规模等）会导致工作流函数对资源需求差异显著，传统资源配置方法无法动态调整，导致资源浪费或性能不足。

本文具体工作如下：

（1）针对无服务器嵌套工作流中的低效执行问题，提出了一种动态函数融合方法。该方法通过启发式算法融合同步调用的函数，减少同步调用的双重计费时间和远程调用的次数，从而显著降低成本和端到端时延。在融合阶段结束后还使用适用于工作流的梯度下降算法优化基础设施配置，确保在实际负载下实现最佳的性能和成本优化。根据此方法设计了一个反馈驱动的函数融合框架，能够根据在线数据动态调整工作流部署实例。最后通过实验验证了所提出的函数融合框架在三个典型嵌套工作流中的有效性，尤其是在减少双重计费、级联冷启动和远程调用开销方面表现突出。实验证明该框架能够快速搜索到函数融合最优解，在成本上分别节约 45.5%、51.9% 和 61.0%，进行基础设施优化后比起函数融合最优解在成本上还能进一步优化 33.9%、41.4% 和 49.3%。

（2）针对无服务器工作流中资源捆绑的分配方法所带来的资源浪费和性能不平衡问题，提出了一种基于资源解耦的自动配置方法。该方法打破了传统无服务器平台中内存与 CPU 资源的紧耦合配置模式，通过独立调整内存和 CPU 资源，能够根据工作流的资源亲和性进行精细化配置，显著降低了资源浪费和运行成本。最后通过实验验证了该方法在三种不同类型的工作流中保证满足 SLO 要求的同时显著降低了资源浪费，比起基线方法 BO 在成本上平均降低了 44.1%、49.9% 和 34.9%，比起基线方

法 MAFF 在成本上平均降低了 31.2%、61.7% 和 45.7%。

（3）针对输入敏感型工作流中不同输入特性对资源需求差异显著的问题，本文提出了一种基于随机森林模型的动态资源配置方法。该方法的核心思想是运用随机森林模型捕捉输入特征与最佳资源配置之间的复杂关系，建立起资源分配预测模型，依据输入特征动态调整资源分配，以应对不同输入对资源需求的显著差异，最大限度地优化资源利用和成本效率。最后通过实验验证了该方法能够根据输入特征自动调整资源配置显著降低成本，针对典型的输入敏感型工作流，该方法能够根据输入特征自动调整资源配置显著降低成本，比起基线方法 BO 在平均成本上降低 45.6%，比起基线方法 MAFF 在平均成本上降低 55.5%。

5.2　后续展望

本文的研究为无服务器工作流的函数融合和资源自动配置提供了理论基础和实践指导，但受限于个人能力和时间，该方向仍有许多方面值得进一步改进：

（1）考虑异构资源的解耦维度：本文提出的基于资源解耦的动态配置方法在优化工作流的内存和 CPU 资源分配方面取得了显著效果。机器学习、深度学习和科学计算等任务通常具有高计算强度和数据密集型特性，传统的 CPU 资源难以满足其性能需求，为了满足开发者对高性能计算的需要，不少无服务器计算平台产生推出了结合 GPU 等异构计算资源的无服务器计算平台。为了进一步支持各类无服务器工作流，未来的研究可以拓展资源解耦配置方法以适配 GPU 等异构计算资源。

（2）函数融合与资源解耦配置的结合：本文提出的动态函数融合框架和基于资源解耦的动态配置框架分别在改善工作流低效执行和优化工作流资源配置方面取得了显著效果。函数融合通过减少函数调用次数和远程调用开销，显著降低了工作流的端到端延迟和成本。然而，函数融合可能会导致融合后的函数资源需求增加，尤其是在融合多个计算密集型函数时，可能会引发资源争用和性能瓶颈。而资源解耦配置通过独立调整内存和 CPU 资源，能够根据工作流的资源亲和性进行精细化配置，显著降低了资源浪费和运行成本。因此，将函数融合与资源解耦配置结合，可以在改善工作流低效执行问题的同时，优化资源分配，进一步提升工作流的性能和成本效益，充分发挥两者的协同效应。

参考文献

[1] SCHMIDT E. Conversation with eric schmidt hosted by danny sullivan[C]//Search Engine Strategies Conference. 2006.

[2] 美国国家网络安全战略[EB/OL]. 2023 [2024-12-01]. https://www.whitehouse.gov/oncd/national-cybersecurity-strategy/.

[3] 数字欧洲工作计划[EB/OL]. 2023 [2024-12-01]. https://digital-strategy.ec.europa.eu/en/news/over-eu760-million-investment-digital-europe-programme-europes-digital-transition-and-cybersecurity.

[4] 国务院关于加快培育和发展战略性新兴产业的决定[EB/OL]. 2010 [2024-12-01]. https://www.gov.cn/zwgk/2010-10/18/content_1724848.htm.

[5] 中央网络安全和信息化委员会印发《"十四五"国家信息化规划》[EB/OL]. 2021 [2024-12-01]. https://www.gov.cn/xinwen/2021-12/28/content_5664872.htm.

[6] SADEEQ M M, ABDULKAREEM N M, ZEEBAREE S R, et al. IoT and Cloud computing issues, challenges and opportunities: A review[J]. Qubahan Academic Journal, 2021, 1(2): 1-7.

[7] SADIKU M N, MUSA S M, MOMOH O D. Cloud computing: opportunities and challenges[J]. IEEE potentials, 2014, 33(1): 34-36.

[8] SANDHU A K. Big data with cloud computing: Discussions and challenges[J]. Big Data Mining and Analytics, 2021, 5(1): 32-40.

[9] LI Y, LIN Y, WANG Y, et al. Serverless computing: state-of-the-art, challenges and opportunities [J]. IEEE Transactions on Services Computing, 2022, 16(2): 1522-1539.

[10] 无服务器计算——AWS Lambda[EB/OL]. 2024 [2024-12-01]. https://aws.amazon.com/cn/lambda.

[11] AWS 上的无服务器[EB/OL]. 2024 [2024-12-01]. https://aws.amazon.com/cn/serverless/.

[12] JONAS E, SCHLEIER-SMITH J, SREEKANTI V, et al. Cloud programming simplified: A berkeley view on serverless computing[J]. arXiv preprint arXiv:1902.03383, 2019.

[13] SHAFIEI H, KHONSARI A, MOUSAVI P. Serverless computing: a survey of opportunities, challenges, and applications[J]. ACM Computing Surveys, 2022, 54(11): 1-32.

[14] EISMANN S, SCHEUNER J, VAN EYK E, et al. A review of serverless use cases and their characteristics[J]. arXiv preprint arXiv:2008.11110, 2020.

[15] YAN M, CASTRO P, CHENG P, et al. Building a chatbot with serverless computing[C]//Proceedings of the 1st International Workshop on Mashups of Things and APIs. 2016: 1-4.

[16] HUSSAIN R F, SALEHI M A, SEMIARI O. Serverless edge computing for green oil and gas industry[C]//2019 IEEE Green Technologies Conference. 2019: 1-4.

[17] ZHANG S, LUO X, LITVINOV E. Serverless computing for cloud-based power grid emergency generation dispatch[J]. International Journal of Electrical Power & Energy Systems, 2021, 124: 106366.

[18] ISHAKIAN V, MUTHUSAMY V, SLOMINSKI A. Serving deep learning models in a serverless platform[C]//2018 IEEE International conference on cloud engineering. 2018: 257-262.

[19] BENEDETTI P, FEMMINELLA M, REALI G, et al. Experimental analysis of the application of serverless computing to IoT platforms[J]. Sensors, 2021, 21(3): 928.

[20] Google Cloud Functions[EB/OL]. 2024 [2024-12-01]. https://cloud.google.com/functions.

[21] IBM Cloud Functions[EB/OL]. 2024 [2024-12-01]. https://cloud.ibm.com/functions.

[22] 阿里云函数计算[EB/OL]. 2024 [2024-12-01]. https://fcnext.console.aliyun.com/.

[23] EISMANN S, SCHEUNER J, VAN EYK E, et al. Serverless applications: Why, when, and how? [J]. IEEE Software, 2020, 38(1): 32-39.

[24] EISMANN S, SCHEUNER J, VAN EYK E, et al. The state of serverless applications: Collection, characterization, and community consensus[J]. IEEE Transactions on Software Engineering, 2021, 48(10): 4152-4166.

[25] AWS Step Functions[EB/OL]. 2024 [2024-12-01]. https://aws.amazon.com/cn/step-functions/.

[26] Azure Durable Functions[EB/OL]. 2024 [2024-12-01]. https://docs.azure.cn/zh-cn/azure-functions/durable.

[27] Google Cloud Workflows[EB/OL]. 2024 [2024-12-01]. https://cloud.google.com/workflows.

[28] MOHAN A, SANE H, DOSHI K, et al. Agile cold starts for scalable serverless[C]//11th USENIX Workshop on Hot Topics in Cloud Computing. 2019: 1-6.

[29] OAKES E, YANG L, HOUCK K, et al. Pipsqueak: Lean lambdas with large libraries[C]//2017 IEEE 37th International Conference on Distributed Computing Systems Workshops. 2017: 395-400.

[30] OAKES E, YANG L, ZHOU D, et al. SOCK: Rapid task provisioning with Serverless-Optimized containers[C]//2018 USENIX annual technical conference. 2018: 57-70.

[31] AKKUS I E, CHEN R, RIMAC I, et al. SAND: Towards High-Performance serverless computing [C]//2018 Usenix Annual Technical Conference. 2018: 923-935.

[32] BERMBACH D, KARAKAYA A S, BUCHHOLZ S. Using application knowledge to reduce cold starts in FaaS services[C]//Proceedings of the 35th annual ACM symposium on applied computing. 2020: 134-143.

[33] DAW N, BELLUR U, KULKARNI P. Xanadu: Mitigating cascading cold starts in serverless function chain deployments[C]//Proceedings of the 21st International Middleware Conference. 2020: 356-370.

[34] MAHGOUB A, WANG L, SHANKAR K, et al. SONIC: Application-aware data passing for chained serverless applications[C]//2021 USENIX Annual Technical Conference. 2021: 285-301.

[35] BARCELONA-PONS D, SÁNCHEZ-ARTIGAS M, PARÍS G, et al. On the faas track: Building stateful distributed applications with serverless architectures[C]//Proceedings of the 20th international middleware conference. 2019: 41-54.

[36] KLIMOVIC A, WANG Y, STUEDI P, et al. Pocket: Elastic ephemeral storage for serverless analytics[C]//13th USENIX Symposium on Operating Systems Design and Implementation. 2018: 427-444.

[37] SHILLAKER S, PIETZUCH P. Faasm: Lightweight isolation for efficient stateful serverless computing[C]//2020 USENIX Annual Technical Conference. 2020: 419-433.

[38] JIA Z, WITCHEL E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices[C]//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 152-166.

[39] ELGAMAL T, SANDUR A, NAHRSTEDT K, et al. Costless: Optimizing cost of serverless computing through function fusion and placement[C]//2018 IEEE/ACM Symposium on Edge Computing. 2018: 300-312.

[40] SCHIRMER T, SCHEUNER J, PFANDZELTER T, et al. Fusionize: Improving serverless application performance through feedback-driven function fusion[C]//2022 IEEE International Conference on Cloud Engineering. 2022: 85-95.

[41] MAHGOUB A, YI E B, SHANKAR K, et al. Wisefuse: Workload characterization and dag transformation for serverless workflows[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2022, 6(2): 1-28.

[42] KHOCHARE A, KHARE T, KULKARNI V, et al. XFaaS: Cross-platform orchestration of FaaS workflows on hybrid clouds[C]//2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing. 2023: 498-512.

[43] SHESHADRI K, LAKSHMI J. RightFusion: Enabling QoS driven Function Fusion in Edge-Cloud FaaS[C]//2024 IEEE International Conference on Cloud Engineering. 2024: 160-167.

[44] AWS Lambda Power Tuning[EB/OL]. 2024 [2024-12-01]. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[45] ZUBKO T, JINDAL A, CHADHA M, et al. Maff: Self-adaptive memory optimization for serverless functions[C]//European Conference on Service-Oriented and Cloud Computing. 2022: 137-154.

[46] AKHTAR N, RAZA A, ISHAKIAN V, et al. COSE: Configuring serverless functions using statistical learning[C]//IEEE INFOCOM 2020-IEEE Conference on Computer Communications. 2020: 129-138.

[47] BILAL M, CANINI M, FONSECA R, et al. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions[C]//Proceedings of European Conference on Computer Systems. 2023: 381-397.

[48] CORDINGLY R, SHU W, LLOYD W J. Predicting performance and cost of serverless computing functions with SAAF[C]//2020 IEEE Intl Conf on Dependable, Autonomic and Secure Comput-

ing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress. 2020: 640-649.

[49] EISMANN S, BUI L, GROHMANN J, et al. Sizeless: Predicting the optimal size of serverless functions[C]//Proceedings of the 22nd International Middleware Conference. 2021: 248-259.

[50] YU G, CHEN P, ZHENG Z, et al. Faasdeliver: Cost-efficient and qos-aware function delivery in computing continuum[J]. IEEE Transactions on Services Computing, 2023, 16(5): 3332-3347.

[51] SAFARYAN G, JINDAL A, CHADHA M, et al. SLAM: SLO-aware memory optimization for serverless applications[C]//2022 IEEE 15th International Conference on Cloud Computing. 2022: 30-39.

[52] RAZA A, AKHTAR N, ISAHAGIAN V, et al. Configuration and placement of serverless applications using statistical learning[J]. IEEE Transactions on Network and Service Management, 2023, 20(2): 1065-1077.

[53] WEN Z, WANG Y, LIU F. Stepconf: Slo-aware dynamic resource configuration for serverless function workflows[C]//IEEE INFOCOM 2022-IEEE Conference on Computer Communications. 2022: 1868-1877.

[54] FENG B, DING Z, ZHOU X, et al. Heterogeneity-aware proactive elastic resource allocation for serverless applications[J]. IEEE Transactions on Services Computing, 2024, 17(5): 2473-2487.

[55] ZHOU Z, ZHANG Y, DELIMITROU C. Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 1-14.

[56] MOGHIMI A, HATTORI J, LI A, et al. Parrotfish: Parametric regression for optimizing serverless functions[C]//Proceedings of the 2023 ACM Symposium on Cloud Computing. 2023: 177-192.

[57] BHASI V M, GUNASEKARAN J R, SHARMA A, et al. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms[C]//Proceedings of the 13th Symposium on Cloud Computing. 2022: 257-272.

[58] SINHA P, KAFFES K, YADWADKAR N J. Shabari: Delayed Decision-Making for Faster and Efficient Serverless Function[J]. arXiv preprint arXiv:2401.08859, 2024.

[59] Docker[EB/OL]. 2024 [2024-12-01]. https://www.docker.com/.

[60] Kubernetes[EB/OL]. 2024 [2024-12-01]. https://kubernetes.io/.

[61] Apache OpenWhisk[EB/OL]. 2024 [2024-12-01]. https://openwhisk.apache.org/.

[62] Knative[EB/OL]. 2024 [2024-12-01]. https://knative.dev/.

[63] AWS Lambda 定价[EB/OL]. 2024 [2024-12-01]. https://aws.amazon.com/cn/lambda/pricing/.

[64] GRAMBOW M, PFANDZELTER T, BURCHARD L, et al. Befaas: An application-centric benchmarking framework for faas platforms[C]//2021 IEEE International Conference on Cloud Engineering. 2021: 1-8.

[65] Serverless TrainTicket[EB/OL]. 2020 [2024-12-01]. https://github.com/FudanSELab/serverless-t

rainticket.

[66] MAHGOUB A, YI E B, SHANKAR K, et al. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs[C] / / Proceedings of Symposium on Network System Design and Implementation. 2022: 303-320.

致 谢

时光荏苒，三年的研究生生涯即将画上句号。回首这段求学之路，我深感幸运与感恩。在此，我谨向所有在我求学过程中给予帮助和支持的老师、同学、家人致以最诚挚的谢意。

首先，感谢我的导师马汝辉老师。感谢蔡子诺博士在我研究过程中的悉心指导。

感谢实验室的同学们：陈星耒、张仁均、王辰霈、金凌啸、刘俊涵。

感谢上海交通大学的同学们：周轶凡、任帅旗、尹明轩、陈泽、朱志柯。

感谢联培的同学们：潘文琪、王佩宇、张建、刘煜丰、卢淑文、夏天、苗义博、刘孟瑾、马千里、曾派、曾铖、姚以真、王煜童。

感谢刘金金和我一起吃喝玩乐，分享生活的乐趣。

最后，我要感谢我的父亲母亲。你们的无私付出和默默支持是我前进的动力。

三年的研究生生涯即将结束，但这段经历将永远铭刻在我的心中。感谢所有帮助过我的人，你们的支持与鼓励让我不断前行。未来的路上，我将继续努力，不负众望。

学术论文和科研成果目录

学术论文

[1] Cai Z, Chen Z, Liu Z, et al. RIDIC: Real-Time Intelligent Transportation System With Dispersed Computing[J]. IEEE Transactions on Intelligent Transportation Systems, 2023, 25(1):1013-1022.

[2] Cai Z, Chen Z, Ma R, et al. SMSS: Stateful Model Serving in Metaverse With Serverless Computing and GPU Sharing[J]. IEEE Journal on Selected Areas in Communications, 2023, 42(3):799-811.

[3] Cai Z, Chen Z, Chen X, et al. SPSC: Stream Processing Framework Atop Serverless Computing for Industrial Big Data[J]. IEEE Transactions on Cybernetics, 2024, 54(11):6509-6517.

专利

[4] 第二发明人，“一种应用于边缘设备的大语言模型流水线推理架构”，专利申请号 CN202411471669.9.

个人简历

基本情况

陈泽彬，男，1998 年 6 月生于广东汕头。

教育背景

- 2022 年 9 月至今，上海交通大学，硕士研究生，计算机技术专业
- 2018 年 9 月至 2022 年 6 月，上海交通大学，本科，机械工程专业

研究兴趣

- Serverless 下的资源自动分配与管理
- Serverless 下的 CPU、GPU 等计算资源的高效共享

联系方式

- 地址：上海市闵行区东川路 800 号，200240
- E-mail: `czb453874483@sjtu.edu.cn`