无服务器计算：挑战，问题和解决方案

陈豪

[1)] 华中科技大学计算机科学与技术学院

摘　要　无服务器计算已成为部署应用程序和服务的新引人注目的范例。它代表了云规划模型、抽象和平台的演变，证明了云技术的成熟和广泛采用。在本文中，我们调查了来自工业界、学术界和开源项目的现有无服务器平台，确定了关键特征和用例，并描述了技术挑战和开放问题。同时，我们介绍了SEUSS[1]，OpenLambda[2]，MPSC[3]，从冷启动、封闭性、多平台协作等方面来解决无服务器计算面临的问题。

关键词　无服务器计算；函数即服务；虚拟化；

Serverless Computing: Challenges, Problems and Solutions

hao chen

[1)] (Department of Computer Science, Huazhong University of Science and Technology, 430000, China)

Abstract

Serverless computing has emerged as a new compelling paradigm for the deployment of applications and services. It represents an evolution of cloud programming models, abstractions, and platforms, and is a testament to the maturity and wide adoption of cloud technologies. In this chapter, we survey existing server- less platforms from industry, academia, and open source projects, identify key characteristics and use cases, and describe technical challenges and open problems. We introduced SEUSS[1], OpenLambda[2] and MPSC[3], which are designed to Solve the problems faced by serverless computing from cold start, closed nature, multi-platform collaboration, etc aspects.

Keywords　Serverless; Function as a Service; Virtualization

1 引言

无服务器计算（也称为功能即服务）[4]正在成为云应用程序部署的一种新的引人注目的范式，这主要是由于最近企业应用程序架构向容器和微服务的转变。

无服务器计算是行业创造的一个术语，用于描述一种编程模型和体系结构，其中小代码片段在云中执行，而无法控制运行代码的资源。这绝不是没有服务器，只是开发人员应该将大多数操作问题（如资源配置、监控、维护、可扩展性和容错）留给云提供商。

精明的读者可能会问，这与平台即服务（PaaS）模型有何不同，后者也抽象了服务器的管理。无服务器模型提供基于无状态函数的“精简”编程模型。与PaaS 不同，开发人员可以编写任意代码，而不限于使用预打包的应用程序。显式使用函数作为部署单元的无服务器版本也称为函数即服务（FaaS）。

从基础架构即服务（IaaS）客户的角度来看，这种范式转变既带来了机遇，也带来了风险。一方面，它为开发人员提供了一个简化的编程模型，用于创建云应用程序，抽象出大多数操作问题; 它通过按执行时间收费而不是资源分配来降低部署云代码的成本; 它是一个平台，用于快速部署响应事件的云原生小段代码。另一方面，在无服务器平台中部署此类应用程序具有挑战性，并且需要放弃平台设计决策，这些决策涉及服务质量（QoS）监视、扩展和容错方案等。

从云提供商的角度来看，无服务器组合提供了额外的机会来控制整个开发堆栈，通过有效优化和管理云资源来降低运营成本。

无服务器平台承诺提供新功能，使编写可扩展的微服务更容易且经济高效，成为云计算架构发展的下一步。包括亚马逊、IBM、微软和谷歌在内的大多数著名的云计算提供商最近都发布了无服务器计算功能。还有一些开源项目，包括OpenLambda[2]项目。

无服务器计算已被用于支持广泛的应用程序。从功能的角度来看，无服务器和更传统的架构可以互换使用。确定何时使用无服务器可能会受到其他非功能性要求的影响，例如对所需操作的控制量、成本以及应用程序工作负载。

从成本角度来看，无服务器架构的好处对于突发、计算密集型工作负载最为明显。突发工作负载表现良好，因为开发人员将函数的弹性交给平台，同样重要的是，函数可以缩放到零，因此当系统空闲时，消费者不会产生任何成本。计算密集型工作负载是合适的，因为在当今的大多数平台中，函数调用的价格与函数的运行时间成正比。因此，受I/O 限制的函数正在为它们未充分利用的计算资源付费。在这种情况下，多路复用请求的多租户服务器应用程序的操作成本可能更低。

从编程模型的角度来看，无服务器函数的无状态性质使其自身适用于类似于函数式响应式编程中的应用程序结构。

2 问题与挑战

无服务器计算已成为部署应用程序和服务的新引人注目的范例。它代表了云规划模型、抽象和平台的演变，证明了云技术的成熟和广泛采用。

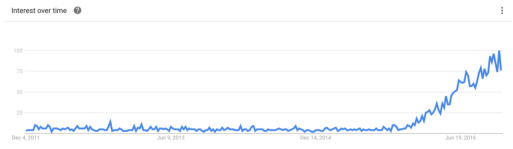

图. 1 Google Trends 报道的“serverless” 受欢迎度

无服务器计算（或简称无服务器）正在成为一种新的、引人注目的云应用程序部署的范例，主要是由于最近企业应用程序架构向容器和微服务的转变。图1显示了Google 趋势报告的“无服务器”搜索词在过去五年中越来越受欢迎。这表明无服务器计算在行业贸易展览、聚会、博客和开发社区中越来越受到关注。相比之下，学术界的关注是有限的。

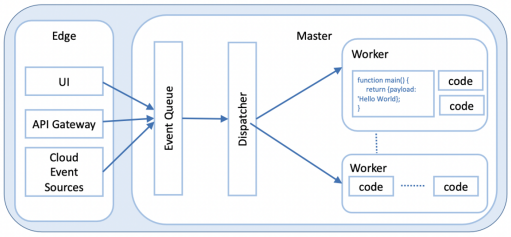

图. 2 无服务平台架构

无服务器平台的核心功能是事件处理系统的核心功能，如图2 所示。服务必须管理一组用户定义的函数，获取通过HTTP 发送或从事件源接收的事件，确定要将事件调度到哪个函数，查找函数的现有实例或创建新实例，将事件发送到函数实例，等待响应，收集执行日志，使响应可供用户使用，并在不再需要函数时停止该函数。

挑战在于实现此类功能，同时考虑成本、可伸缩性和容错等指标。平台必须快速有效地启动函数并处理其输入。平台还需要对事件进行排队，并根据队列的状态和事件的到达率，调度函数的执行，管理空闲函数实例的停止和解除分配资源。此外，

平台需要仔细考虑如何在云环境中扩展和管理故障[5]。

2.1 无服务平台调研

在本节中，我们将比较许多无服务器平台。我们首先列出将用于表征这些平台架构的维度，然后简要描述每个平台。

2.1.1 特征

有许多特征有助于区分各种无服务器平台。开发人员在选择平台形式时应注意这些属性。

- 成本：通常按使用量计量，用户只需为运行无服务器函数时使用的时间和资源付费。这种缩放到零实例的能力是无服务器平台的关键区别之一。按流量计费的资源（如内存或CPU）和定价模型（如非高峰期计数）因提供商而异。
- 性能和限制：对无服务器代码的运行时资源要求设置了各种限制，包括并发请求数以及可用于函数调用的最大内存和CPU 资源。当用户的需求增长时，可能会增加一些限制，例如并发重新请求阈值，而其他限制则是平台固有的，例如最大内存大小。
- 编程语言：无服务器服务支持多种编程语言，包括Javascript，Java，Python，Go，C#和Swift。大多数平台形式支持一种以上的编程语言。某些平台还支持以任何语言编写的代码的扩展性机制，只要将其打包在支持明确定义的API的Docker 映像中即可。
- 编程模型：目前，无服务器平台通常执行单个主函数，该函数将字典（例如JSON对象）作为输入并生成字典作为输出。
- 可组合性：平台通常提供某种方式来从一个无服务器函数调用另一个无服务器函数，但某些平台提供了更高级别的机制来组合这些函数，并且可能使构建更复杂的无服务器应用变得更加容易。
- 部署：平台努力使部署尽可能简单。通常，开发人员只需要提供一个包含函数源代码的文件。除此之外，还有许多选项可以将代码打包为包含多个文件的存档，或者打包为带有二进制代码的Docker 映像。同样，版本化或分组函数的功能也很有用，但很少见。
- 安全性和记帐：无服务器平台是多租户的，必须隔离用户之间的功能执行，并提供详细的记帐，以便用户了解他们需要支付多少费用。
- 监视和调试：每个平台都支持使用执行日志中记录的print 语句进行基本调试。可以提供其他功能来帮助开发人员发现瓶颈、跟踪错误并更好地消除功能执行的流程。

2.1.2 商业平台

亚马逊的AWS Lambda是第一个无服务器平台，它定义了几个关键维度，包括成本、编程模型、部署、资源限制、安全性和监控。支持的语言包括Node.js、Java、Python 和C#。初始版本的可组合性有限，但最近已解决此问题。该平台利用了大型AWS 服务生态系统，可以轻松地将Lambda 函数用作事件处理程序，并在编写服务时提供粘附代码。

Google Cloud Functions 目前作为Alpha 版本提供，提供基本的FaaS 功能来运行用Node 编写的无服务器函数以响应来自某些Google Cloud 服务的HTTP 调用或事件。该功能目前受到限制，但预计在未来版本中会增长。

Microsoft Azure Functions 提供HTTP webhook 以及与Azure 服务的集成，以运行用户提供的函数。该平台支持C#，F#，Node.js，Python，PHP，bash或任何可执行文件。运行时代码是开源的，可在GitHub 上使用MIT 许可证。为了简化调试，Azure 功能CLI 提供了用于创建、开发、测试、运行和调试Azure Functions 的本地开发体验。

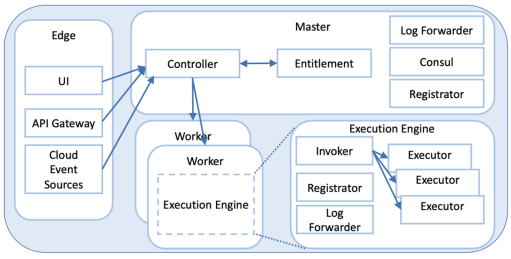

图. 3 IBM OpenWhisk 架构

IBM OpenWhisk 提供基于事件的无服务器编程，能够链接无服务器函数以创建复合函数。它支持Node.js，Java，Swift，Python以及嵌入Docker容器中的任意二进制文件。OpenWhisk 在GitHub 上以Apache 开源许可证提供。OpenWhisk平台的主要架构组件如图3所示。与图2 中的通用架构图相比，我们可以看到还有其他组件处理重要需求，如安全性、日志记录和监视。

2.1.3 新无服务器平台

OpenLambda[2]是一个开源的无服务器计算平台。源代码在GitHub 中以Apache 许可证提

供。OpenLambda论文概述了围绕性能的许多挑战，例如支持异构语言运行时和跨负载平衡服务器池的更快功能启动时间，部署大量代码，在无状态功能之上支持有状态交互（例如HTTP会话），将无服务器功能与数据库和数据聚合器一起使用，遗留分解和成本调试。我们在第3 节中确定了类似的挑战。

一些无服务器系统是由公司创建的，它们看到在其运营的环境中需要无服务器计算。例如，Galactic Fog将无服务器计算添加到他们在Mesos D/C 之上运行的格式塔框架中。源代码在Apache 2 许可证下可用。Auth0 创建了执行无服务器功能的webtasks，以支持复杂安全场景中使用的webhook 端点。此代码也可作为开源代码使用。自2012 年以来，Iron.io 对任务提供了无服务器支持。最近他们宣布了Project Kratos，允许开发人员将AWS Lambda 函数转换为Docker 图像，并且在Apache 2 许可下可用。此外，他们还与Cloud Foundry 合作，为Cloud Foundry 用户提供多云无服务器支持。LeverOS 是一个开源项目，它使用RPC 模型在服务之间进行通信。LeverOS 中的计算资源可以被标记，因此重复的函数调用可以针对特定容器以优化运行时性能，例如利用容器中的热缓存。

2.2 优势和缺陷

与IaaS 平台相比，无服务器架构在控制、成本和灵活性方面提供了不同的权衡。特别是，它们迫使应用程序开发人员在模块化应用程序时仔细考虑其代码的成本，而不是延迟、可伸缩性和弹性，而延迟、可伸缩性和弹性是传统上花费大量开发工作的地方。

无服务器范例对使用者和提供者都有优势。从消费者的角度来看，云开发人员不再需要配置和管理服务器、虚拟机或容器作为提供分布式服务的基本计算构建块。相反，重点是业务逻辑，通过定义一组函数，这些函数的组合支持所需的应用程序行为。无状态编程模型使提供商能够更好地控制软件堆栈，从而使他们能够更透明地提供安全补丁并优化平台。

然而，消费者和提供者都有缺点。对于消费者来说，该平台提供的FaaS模型对于某些应用来说可能过于局限。例如，平台可能不支持最新的Python 版本，或者证书库可能不可用。对于提供商来说，现在需要管理诸如用户功能的生命周期、可扩展性和容错等问题以与应用程序无关的方式。这也意味着开发人员必须仔细了解平台的行为方式，并围绕这些功能设计应用程序。

无服务器平台的一个属性在一开始可能并不明显，那就是提供商倾向于提供一个服务生态系统来增强用户的功能。例如，可能有用于管理状态、记录和监视日志、发送警报、触发事件或执行身份验证和授权的服务。如此丰富的生态系统可以吸引开发人员，并为云提供商提供另一个收入机会。然而，使用此类服务会带来对提供商生态系统的依赖，以及供应商锁定的风险。

2.3 挑战和问题

我们将列出根据我们使用无服务器服务的经验已知的挑战，然后描述未解决的问题。

2.3.1 系统级挑战

- 成本：成本是一个根本性的挑战。这包括最小化无服务器函数的资源使用量，无论是在执行时还是在空闲时。另一个方面是定价模型，包括它与其他云计算方法的比较。例如，无服务器函数目前对于CPU 密集型计算最经济，而I/O 密集型函数在专用VM 或容器上可能更便宜。

- 冷启动：无服务器的一个关键区别在于能够缩放到零，或者不向客户收取空闲时间费用。但是，缩放到零会导致冷启动的问题，并付出让无服务器代码准备好运行的代价。在将冷启动问题最小化的同时仍缩放到零的技术至关重要。

- 资源限制：需要资源限制来确保平台可以应对负载峰值并管理攻击。无服务器函数的可强制资源限制包括内存、执行时间、带宽和CPU 使用率。此外，还有可应用于多个函数或整个平台的总资源限制。

- 安全性：功能的强大隔离至关重要，因为来自许多用户的函数在共享平台上运行。

- 扩展：平台必须确保用户函数的可扩展性和弹性。这包括主动预配资源以响应负载和预测未来的负载。在无服务器中，这是一个更具挑战性的问题，因为这些预测和预配决策必须在很少或根本没有应用程序级知识的情况下做出。例如，系统可以使用请求队列长度作为负载的指示，但对这些请求的性质视而不见。

- 混合云：随着无服务器越来越受欢迎，可能需要协同工作的无服务器平台和多个无服务器服务。一个平台不太可能拥有所有功能并适用于所有用例。

- 遗留系统：从无服务器平台中运行的无服务器代码访问较旧的云和非云系统应该很容易。

2.3.2 编程模型和开发运营挑战

- 工具：如果访问服务器以便能够监视和调试应用程序的传统工具不适用于无服务器架构，需要新的应用程序。

- 部署：开发人员应该能够使用声明性方法来控制部署的内容和支持它的工具。

- 监视和调试：由于开发人员不再拥有可以访问的服务器，因此无服务器服务和工具需要关注开发人员的工作效率。由于无服务器函数运行的时间较短，因此运行它们的数量级将增加许多，从而更难识别问题和瓶颈。当函数完成时，其执行的唯一跟踪是无服务器平台监视记录的内容。

- IDE：需要更高级别的开发人员功能，例如重构功能（例如，拆分和合并功能），恢复到旧版本等，并且应该与无服务器平台完全集成。

- 可组合性：这包括能够从一个函数调用另一个函数，创建调用和协调许多其他函数的函数，以及更高级别的构造，如并行执行和图形。将需要工具来促进工作的创建及其维护。

- 长时间运行：目前无服务器函数的执行时间通常受到限制。有些方案需要长时间运行（如果间歇性）逻辑。编程模型和工具可以将长时间运行的任务分解为较小的单元，并提供必要的上下文来将它们作为一个长时间运行的工作单元进行跟踪。

- 状态：实际应用程序通常需要状态，并且不清楚如何在无状态无服务器函数中管理状态-编程模型、工具、库等需要提供必要的抽象级别。

- 并发性：表达并发语义，如原子性（函数执行需要序列化）等。

- 恢复语义：只包括一次、最多一次和至少一次语义。

- 代码粒度：目前，无服务器平台以函数的粒度封装代码。更粗或更细的模块是否有用是一个悬而未决的问题。

2.3.3 开放研究问题

现在我们将描述一些未解决的问题。我们将它们框定为问题，以强调它们在很大程度上是未开发的研究领域。

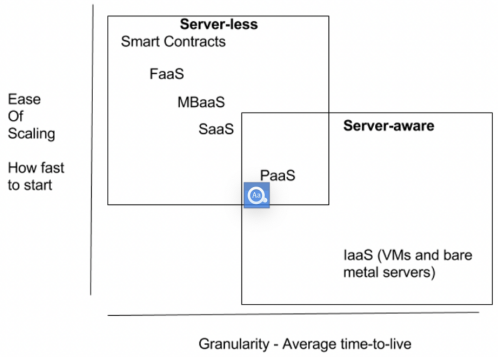

图. 4 该图显示了生存时间（x 轴）和易于缩放（y 轴）之间的关系。服务器感知计算（裸机、虚拟机、IaaS）有很长的生命周期并且需要更长的时间来扩展（提供新资源的时间）；无服务器计算（FaaS、MBaaS、PaaS、SaaS）经过优化，可在多台服务器上运行并隐藏服务器详细信息。

无服务器的边界是什么？关于无服务器计算的一个基本问题是边界：它是仅限于FaaS还是范围更广？它与其他模型（如SaaS和MBaaS）有何关系？

随着无服务器的普及，不同类型的“即服务”之间的界限可能正在消失（参考图4）。可以想象，开发人员不仅编写代码，而且还声明他们希望代码如何运行- 如FaaS或MBaaS或PaaS - 并且可以随着需求的变化而改变。将来，主要区别可能是关心服务器（服务器感知）和不关心服务器细节（无服务器）。PaaS 位于中间; 它使部署代码变得非常容易，但开发人员仍然需要了解服务器并了解扩展策略，例如要运行的实例数。

不同的云计算服务模式可以混合使用吗？对于无服务器函数可以使用多少内存和CPU，是否可以有更多选择？无服务器是否需要具有类似IaaS 的定价？具有动态变化粒度的现场和动态筛选怎么样？

无服务器工具与现有解决方案有根本区别吗？由于无服务器的粒度比传统的基于服务器的工具小得多，我们可能需要新的工具来很好地处理数量更多但寿命更短的架构。监视和调试无服务器应用程序将更具挑战性，因为没有服务器可以直接访问以查看出了什么问题。相反，无服务器平台需要在代码运行时收集所有数据，并在以后使其可用。同样，如果开发人员需要处理无数较小的代码片段，而不是只有一个工件（微服务或传统的整体应用程序），则调试也大不相同。可能需要新的方法来虚拟地将无服务器部分模拟成更易于理解和推理的更大单元。

是否可以使旧代码运行无服务器？必须继续运行的现有（“遗留”）代码的数量远远大于专门为在

无服务器环境中运行而创建的新代码。现有代码的经济价值代表了开发人员编码和修复软件的无数小时的巨大投资。因此，最重要的问题之一可能是现有的遗留代码可以在多大程度上自动或半自动分解为更小粒度的部分，以利用这些新的资源。

无服务器从根本上是无状态的吗？由于当前的无服务器平台是无状态的，因此将来会有有状态的无服务器服务吗？会有简单的方法来处理状态吗？不仅如此：无服务器从根本上是无状态的吗？是否有无服务器服务内置了具有不同程度服务质量的有状态支持？

是否有构建无服务器解决方案的模式？如何将无服务器的低粒度基本构建块组合到更大的解决方案中？我们将如何将应用程序分解为函数，以便它们优化资源使用？例如，我们如何识别构建为在无服务器服务中运行的应用程序的CPU 密集型部分？我们可以使用定义良好的模式来组合函数和外部API 吗？在服务器和客户端上应该做什么（例如，重客户端在这里更合适）？是否有可以从OOP 设计模式、企业集成模式等中应用的经验教训？

无服务器是否超越了传统的云平台？无服务器可能需要支持在传统定义的数据中心之外执行代码的方案。这可能包括将云扩展到包括物联网、移动设备、Web 浏览器和边缘的其他计算的努力。例如，“雾”计算的目标是创建一个系统级水平架构，将计算，存储，控制和网络的资源和服务分布在从云到物联网的序列中。在“雾”和云之外运行的代码不仅可以嵌入，还可以虚拟化，以允许在设备和云之间移动。

另一个例子是运行在区块链中执行“智能合约”编排交易的代码。定义合约的代码可以部署和运行在超级账本结构对等节点网络上，也可以在以太坊对等网络的任何节点上的以太坊虚拟机中部署和运行。由于系统是去中心化的，因此没有以太坊服务或服务器来运行无服务器代码。相反，为了激励以太坊用户运行智能合约，他们从代码消耗的能量中获得报酬，类似于汽车的燃料成本，但适用于计算。

3 OpenLambda

[2]介绍了OpenLambda，这是一个新的开源平台，用于在新兴的无服务器计算模型中构建下一代Web服务和应用程序。[2]描述了无服务器计算的关键方面，并提出了在设计和实施此类系统时必须解决的许多研究挑战。[2]还简要介绍了当前的Web应用程序，以便更好地激励无服务器应用程序构建的某些方面。

数据中心及其中的软件平台的快速创新步伐将再次改变我们构建、部署和管理在线应用程序和服务的方式。在早期设置中，每个应用程序都在自己的物理机器上运行。购买和维护大量机器的高成本，以及每台机器往往未得到充分利用的事实，导致了一个巨大的飞跃：虚拟化。虚拟化支持将服务大量整合到服务器上，从而大大降低了成本并提高了可管理性。

然而，基于硬件的虚拟化并不是灵丹妙药，更轻量级的技术已经出现来解决其基本问题。这个领域的一个主要解决方案是容器，一种面向服务器的Unix风格进程的重新打包，并增加了命名空间虚拟化。结合Docker 等分发工具，容器使开发人员能够轻松启动新服务，而无需虚拟机和运行时开销。

基于硬件和基于容器的虚拟化的共同点是服务器的核心概念。服务器长期以来一直被用来支持在线应用程序，但新的云计算平台预示着传统后端服务器的终结。众所周知，服务器难以配置和管理，服务器启动时间严重限制了应用程序快速扩展和缩减的能力。

因此，一种称为无服务器计算的新模型有望改变现代可扩展应用程序的构建。开发人员没有将应用程序视为服务器的集合，而是使用一组可以访问公共数据存储的函数来定义应用程序。这个微服务的一个很好的基于平台的例子可在亚马逊的Lambda 中找到; 因此，我们通常将这种服务构造风格称为Lambda 模型。

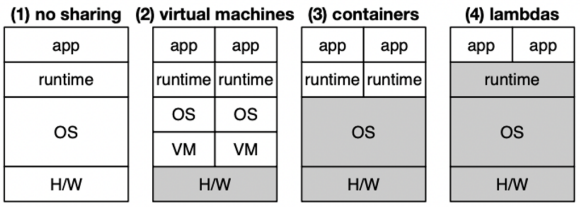

图. 5　共享的演化（灰色为共享的部分）

与更传统的基于服务器的方法相比，Lambda 模型具有许多优势。来自不同客户的Lambda 处理程序共享由云提供商管理的通用服务器池，因此开发人员不必担心服务器管理。处理程序通常用JavaScript 或Python 等语言编写; 通过跨功能共享运行时环境，特定于应用程序的代码通常很少，因此将处理程序代码发送给集群中的任何工作线程的成本很低。最后，应用程序可以快速扩展，而无需启动新服务器。通过这种方式，Lambda 模型代表了应用程序之间共享演变的逻辑，从硬件到操作系统，再到运行时环境本身（图5）。

在[2]中，作者提出了Lambda模型并提出了相关的研究挑战。Lambda 执行引擎必须安全有效

地隔离处理程序。处理程序本质上是无状态的，因此Lambda 和数据库服务之间有很多集成的机会。Lambda 负载均衡器必须在考虑会话、代码和数据局部性时做出低延迟决策。进一步探讨了即时编译，包管理，Web会话，数据聚合，货币成本和可移植性的新挑战。不幸的是，大多数现有的实现（OpenWhisk 除外）都是封闭和专有的。为了促进对Lambda 架构的研究，作者构建了OpenLambda，这是一个研究者可以评估无服务器计算新方法的基础。

我们已经看到，Lambda 模型比以前的平台更具弹性和可扩展性，包括基于容器的自动扩展服务。我们还看到，这种新范式为执行引擎、数据库、调度程序和其他系统提供了有趣的方法。为了促进这些领域的研究，[2]正在构建OpenLambda，这是Lambda 平台的开源版本。OpenLambda 将由许多子系统组成，这些子系统将协调运行Lambda 处理程序：用于托管和分发处理程序代码的Lambda 存储，用于沙箱处理程序的本地执行引擎，用于在worker 之间分发请求的负载平衡器，以及Lambda 感知分布式数据库。[2]进一步计划构建LambdaBench，这是一个新的基准测试套件，基于各种应用程序的端口到Lambda 编程模型。作者希望提供构成Lambda 基础设施的所有组件的完整集合，将使研究人员能够评估无服务器计算平台内各种子系统的新设计和实现。

4 MPSC

随着FaaS 技术的快速采用和众多自托管FaaS 系统的引入，在提供商生态系统中对功能的实时监控和调度的需求变得至关重要。在[3]中，作者介绍了MPSC，一个支持多提供商无服务器计算的框架。MPSC 实时监控无服务器提供程序的性能，并跨这些提供商调度应用程序。此外，MPSC 还提供API供用户定义自己的调度算法。与在单一的云资源上进行调度相比，MPSC 在易变的边缘计算环境中跨多个提供商提供了4 倍的加速。

在过去的二十年中，云计算为用户提供了近乎实时地按需配置资源的能力，并消除了购买硬件所涉及的资本支出，从而彻底改变了IT 行业。基础设施硬件的虚拟化使云计算能够为大量的开发人员提供弹性计算和无限容量的错觉。在容器技术的支持下，无服务器计算进一步完善了计算资源的细分、弹性和容器首创的重点硬件的抽象。无服务器技术（也称为函数即服务）已被证明比以前的云架构更优秀。FaaS 技术允许开发人员使用各种共享运行时环境发布微服务（称为函数或操作），而无需管理单个服务器。每个函数代表一个小的无状态代码片段，可以通过事件触发。事件通常来源于HTTP请求、数据库变化，或者另一个函数。由于无状态函数的性质，它们不易并行化、水平扩展、充分利用硬件资源的全部功能。

无服务器技术将所有服务器管理和函数执行转移到服务提供商，从而实现高效的自动扩容和快速的市场发展。无服务器计算使开发人员能够专注于其代码，从而将他们从虚拟机和容器中看到的服务器和部署详细信息中解放出来。FaaS 提供的更高抽象级别最大限度地减少了开发人员的样板代码和扩展系统的配置管理。通过HTTP 接口解耦功能，无服务器架构加强了模块化开发。模块化开发允许组件之间的明确分离和功能隔离，这两者都被广泛认为是软件设计中的最佳实践，从而提高软件质量、测试能力和生产力。

Provider	Runtime	Classification
Amazon	Lambda	commercial
Microsoft	Azure Functions	commercial
Google	Cloud Functions	commercial/beta
IBM	Apache OpenWhisk	open-source
Host Agnostic	Kubeless	open-source
	Fission	open-source
	OpenFaas	open-source
	Nuclio	open-source
	FnProject	open-source

图. 6　提供商和运行时

无服务器技术越来越受欢迎，新提供商迅速涌现。与主机无关的开源产品的介绍如图6所示。随着新提供商的出现，客户可以选择在哪里运行他们的代码。每个提供商都有独特的功能，并提供多种配套服务，即监控、日志记录和数据库。此外，每个提供商都有独特的性能特征，并且容易受到中断的影响。性能特征和服务中断可能由多种原因引起，包括托管位置或FaaS 基础架构的准备实施。

人们可能会想象一个生态系统，开发人员只需编写一次代码即可部署到任何FaaS 提供商上。事实正好相反，供应商需要独一无二的函数签名、命名规范和事件兼容性，导致了供应商锁定效应。幸运的是，两个开源框架已经加入进来，帮助互连无服务器提供商，Serverless Framework 和Event Gateway。Serverless Framework 无服务器框架简化和标准化了功能开发，允许以最少的代码更改在所有主要的FaaS 提供商上进行部署。Event Gateway 标准化了事件，允许跨提供商通信。

即使Serverless Framework 和Event Gateway的出现为跨供应商的利用奠定了基础，但迄今为止，在具有独特性能特征的供应商生态系统中，还没有用于调度功能的基础结构。为了帮助选择“正确”的

提供商，作者开发了MPSC框架，这是一个无服务器监控和调度系统。MPSC 首先实时监控无服务器提供商的性能。其次，基于性能结果和用户定义的调度，MPSC 可以推荐运行用户代码的最佳位置。MPSC 允许用户根据提供商的动态平均执行时间、资源亲和力和提供商成本实施调度算法。利用MPSC 的自适应调度已被证明可以提高性能和QoS。

[3]设计并实施了MPSC，这是一个用于支持多提供商无服务器计算的框架。MPSC 旨在为用户提供一个平台，以便根据可自定义的调度算法跨多个无服务器提供商调度其应用程序。

4.1 系统设计

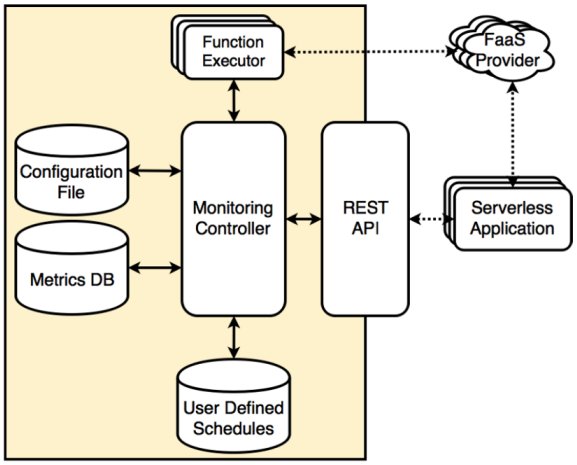

图. 7　MPSC 架构

MPSC 架构由松散耦合的微服务、利用REST风格的通信和多个可扩展的可插拔组件组成。MPSC 旨在实现最大的灵活性，支持创建自定义调度逻辑和执行程序，以便与新的FaaS 提供商一起使用。此外，通过允许通过标准HTTP 进行通信，应用程序可以轻松地容器化并部署在多种环境中。MPSC 架构如图7 所示，下面描述了组件的更多详细信息。

- 监控控制器是MPSC 框架的关键组件，负责将模型对象（配置文件、指标数据库和用户定义的调度）数据与函数执行器和REST API 进行协调。
- REST API 为用户及其应用程序提供了一系列HTTP 端点，以便与MPSC 框架进行通信。当前版本下的四个端点为：指标，测试，添加调度，调度。
- 函数执行器是一组可扩展的互连，用于调用和测量FaaS 提供程序上的功能性能。目前，有两个函数执行器，一个用于处理AWS Lambda 上的函数，另一个用于处理OpenWhisk 上的函数。
- 配置文件包含每个提供程序的用户特定配置属性。
- 度量数据库存储配置文件中指定的每个提供程序的基准结果。数据库中的每个条目都包含所调用函数的名称、往返时间和错误状态。
- 用户定义调度通过RESR API 上传python 的调度。每个被提交的调度必须是ComputeBaseClass 的子类，并且实现了调度方法。

4.2 工作流

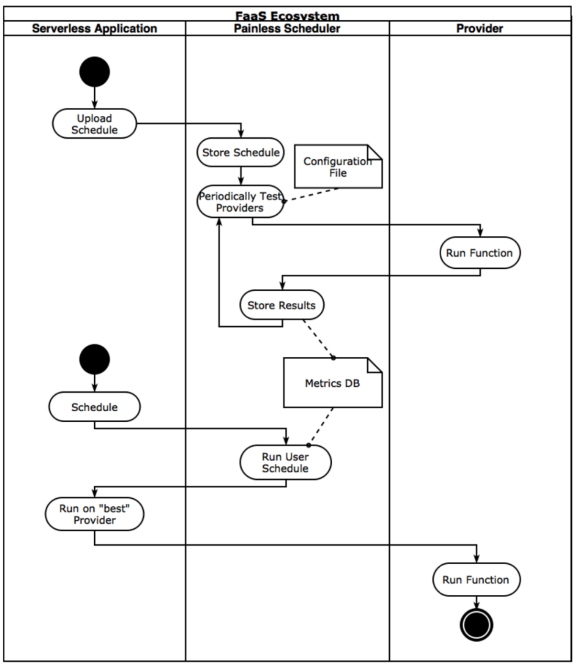

图. 8　MPSC 工作流

MPSC 框架是一个无服务器监控和调度系统，旨在为无服务器应用程序实现更高的QoS。图8 显示了MPSC 框架的基本工作流，其中用户上传计划并查询系统以确定用于其无服务器应用程序的提供程序。作为此工作流的结果，无服务器应用程序在由其计划算法确定的最合适的提供程序上运行。运行图8 中所示的工作流有两个先决条件：完成配置文件和编写用户定义的计划。

配置文件在MPSC 系统启动时加载，包括与每个FaaS 提供程序相关的用户特定信息。由于配

置文件仅在系统启动时加载，因此请务必注意，在重新启动之前，不会识别对文件的任何更改。在YAML 格式的文件中，用户必须为要监视/计划的每个提供程序指定以下属性。

- Provider Type - 提供商类型表示为compute：providers键值对。目前，支持AWS或OpenWhisk的价值。
- ApiHost – 提供商使用的域名
- Namespace -命名空间或者用户的域名
- Cost – 每次请求的花费（美元）

4.3 用户定义调度

MPSC 为用户提供了定义自己的调度算法的灵活性。用户定义的调度是通过对调度程序包中的ComputeBaseClass 进行子类化来实现的。在子类实现中，需要实现调度方法，并使用提供程序的name 方法返回所选提供程序的名称。此外，请务必注意，在实施自定义计划时，用户目前可以使用三个指标。每个提供商包含的三个指标是平均往返时间、导致错误的请求数和每次调用的成本。通过利用调度插件接口，用户可以根据其独特的无服务器需求自定义和定制MPSC。例如，每次调用成本低于0.01 USD 且平均往返时间小于20 毫秒的提供商。

5 SEUSS

本章介绍了一种在FaaS 环境中实现无服务器功能的快速部署和高密度缓存的系统级方法SEUSS[1]。为了缩短启动时间，从单内核快照部署函数，绕过昂贵的初始化步骤。为了减少快照的内存占用，我们在运行函数所需的整个软件堆栈中应用页面级共享。我们通过在FaaS 平台架构的计算节点上替换Linux 来演示我们技术的效果。使用这个原型操作系统，函数的部署时间从100 毫秒缩短到10 毫秒以下。完全由新功能组成的工作负载的平台吞吐量提高了51 倍。该系统能够在内存中缓存超过50000 个函数实例，而不是使用标准操作系统技术缓存3000 个。这些改进相结合，使FaaS 平台具有处理大规模突发请求的新能力。

在功能即服务（FaaS）模型中，无服务器函数是为高级语言解释器（如Node.js，Python或Java）编写的简短代码段。在此模型中，函数由远程FaaS 平台执行以响应调用请求。无服务器功能促进应用程序设计，这些设计由许多短期执行组成，这些执行以单例的形式快速部署，按顺序或并行部署。对于程序员来说，无服务器函数为跨任意规模访问按需计算提供了强大的原语支持。对于FaaS平台，从应用程序开发人员的操作转移允许系统控制功能的部署方式。例如，功能可以部署在专用容器、虚拟机、进程级封装或语言级ISO 中。

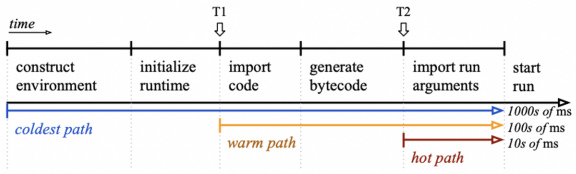

图. 9　函数触发的阶段

引导解释器和编译源代码可能会导致函数部署的长时间初始化开销。因此，为了实现低延迟启动时间，FaaS 平台必须在函数的调用生命周期中缓存中间阶段以供重用（图9）。为了实现许多不同功能的快速部署，需要FaaS 平台在内存中保存大量缓存的状态。例如，通过假设一小组解释器，平台可以管理正在运行的解释器的预初始化池（图9 中的T1）。此外，函数的无状态特性使平台能够缓存特定函数的执行环境（例如，运行函数的容器或VM），以便在将来的调用中立即重用（图9 中的T2）。在内存中缓存许多隔离环境的要求，其中许多环境在重用方面仅限于单个函数，这提出了现代操作系统处理缓存的独特挑战。

5.1 SEUSS 方法

SEUSS 是一种系统级方法，用于在多租户FaaS 环境中快速部署无服务器功能。其目标是为无服务器函数提供通用解决方案，该解决方案支持各种语言运行时，隔离执行函数，并有效地缓存应用程序和系统状态以实现快速部署。我们通过从单核快照部署函数来实现这些所需的属性。

5.1.1 单核上下文(UCs)

[1]的关键见解是，单核化行为将类似进程的计算转换为适合计算缓存的格式。单核提供了一种机制，我们用它来封装和观察无服务器函数的执行状态，无论它在软件堆栈中出现在哪里。

在单内核中，应用程序及其支持的系统功能（例如，文件系统、用户库和网络堆栈）被组合到单个地址空间中。在SEUSS中，每个单核上下文（UC）都由一个高级语言解释器（例如，Node.js，Python）组成，该解释器配置为导入和执行函数代码。UC 充当我们单独隔离的函数执行的部署单元。

单核化的明显缺点是，它可以通过复制曾经共享的状态和函数来引入大量复制的状态和冗余初始化过程。但是，可以通过使用快照、快照堆栈和预

期优化来检测和减少这些冗余。

5.1.2 快照

快照是一个不可变的数据对象，它表示UC的瞬时执行状态（即其地址空间和寄存器）。在SEUSS 中，快照充当可以从中部署UC 的模板。由于快照是不可变的内存映像，因此可以同时随着时间的推移从单个快照启动任意数量的UC。通过在基本快照中捕获完全初始化的UC，每个部署的UC 都可以避免启动单内核和设置解释器的开销。

将执行封装在UC 中可以最大程度地减少UC外部的执行状态量。因此，快照和从快照部署的操作是通过其后备数据结构对地址空间进行简单的操作。此外，当快照可以在外部进行触发和收集时，捕获UC的行为不需要对UC的内部状态进行语义理解。换句话说，快照可以将UC中的软件视为黑匣子。我们的方法与基于fork的方法形成鲜明对比，后者需要明确的应用程序级支持和与内核的协调。例如，快照方法在不同的内部处理器中的工作方式相同，包括本身不支持fork的解释器（例如Node.js）。

5.1.3 快照堆栈

快照堆栈表示快照之间随时间变化的世系，就像分叉树捕获进程的沿袭关系一样。快照堆栈的工作原理是将每个快照视为快照堆栈中上一个快照的页面级差异。为此，我们使用传统的写入时复制语义将目标UC 修改的页面仅捕获到快照中。

快照堆栈旨在通过分解来增加可在内存中缓存的函数数量，在UC 之间共享的通用执行状态。为了更好地发挥快照堆栈的优势，请考虑以下示例：FaaS 平台希望对JavaScript 函数Foo 和Bar 的完全初始化状态进行快照。仅使用快照机制，该平台需要两个UC 快照，每个功能一个。如果输入是100MB，每个函数增加1MB，我们需要202 MB的存储空间。对于快照堆栈，使用三个快照，一个用于初始化的JavaScript 解释器，第二个用于Foo diff，第三个用于Bar diff。这需要102 MB，因为解释器在两个函数快照之间共享。

5.1.4 预期优化

我们将预期优化（AO）定义为在捕获快照之前有意运行计算的行为，目的是从后续执行中减少多余的空间和时间使用。基于相同托管运行时的函数将执行类似的路径来设置和执行函数。AO 提供了进一步将初始化过程和内存迁移到共享快照中的机会。我们针对的冗余包括解释器内部动态生成的数据结构或单核子系统，这需要大量精力手动优化。

在SEUSS 中，我们通过在捕获基本运行时快照之前先发制人地预热UC 软件堆栈的内部路径和数据结构来应用AO。AO通过启用从快照生成的执行来避免昂贵的分配和启动过程，从而减少了启动和执行时间。此外，该技术通过减少每个快照中捕获的写入页数来提高特定于函数的快照的可缓存性。使用AO，基本快照的内存占用量会增加，但这种空间权衡会导致性能提高，并减少从基本快照中生成的所有快照和UC 实例的占用空间。

我们在将AO 应用于单内核环境方面的经验表明，它为实施重大性能优化提供了一个直观的工具。我们使用有关部署高级函数代码任务的语义知识来预测和预执行常见过程。例如，我们探讨了将TCP流量发送到单内核的好处，并在捕获基本快照之前通过JavaScript解释器运行（虚拟）脚本。这两个AO 导致暖函数和冷函数执行时间成倍减少（如图9所示），以及加倍快照缓存密度。重要的是，我们使用的AO是通过对导入和部署函数代码的高级过程的基本推理发现的，而不了解或关心解释器和内核的微妙内部工作。

5.2 SEUSS 操作

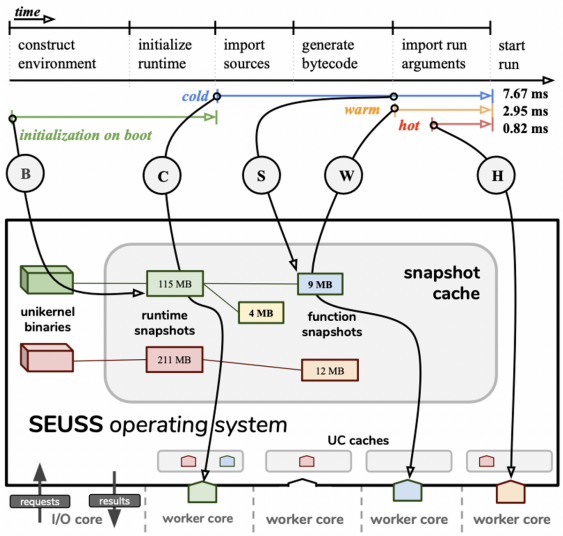

图. **10** 部署在Faas 节点上的SEUSS 系统的高阶操作

本节介绍在FaaS 计算节点上使用SEUSS 部署无服务器函数的高级过程，如图10所示。

作为早期系统初始化的一部分，单内核被引导到语言解释器中，并运行一个调用驱动程序（脚本），该驱动程序创建HTTP/REST 端点。系统通过此脚本将命令和函数代码发送到单内核中。在调用驱动程序启动后拍摄运行时快照（图10中

的B）。这些运行时快照的内存使用量可能相对较大（数百MB），但数量很少：每个受支持的解释器只有一个。从快照部署新UC 时，内部驱动程序将在侦听状态下启动，准备接受新连接。

为了部署无服务器函数，SEUSS 维护快照缓存以及空闲UC 缓存。从远程FaaS 平台控制器接收函数调用请求，之后系统可以采用以下三种路径之一来处理每个调用：冷、温或热。当要调用的特定函数不存在缓存快照时，将从基本运行时快照部署其UC，并将函数源导入到UC 中并由运行时解释（图10中的C）。完成函数源编译步骤后，将捕获特定于函数的快照（图10中的S）。接下来，将运行参数导入到UC 中，并开始执行函数，完成冷路径调用。

当调用的函数存在特定于函数的快照时，通过从该快照创建UC、跳过代码导入和编译阶段、导入运行参数，最后开始函数执行来采用暖路径（图10中的W）。函数执行完成后，可以销毁或缓存其UC，以便将来在一组新参数上调用该函数。热调用路径包括将一组新的运行参数导入到已构造的UC 中（图10中的H）。

5.3 安全性

安全性必须是多租户云环境中的核心设计问题。在SEUSS 中，单内核用于将共同运行的函数实例彼此隔离，并与受信任的内核隔离。我们的原型使用标准x86 用户/内核保护域将不受信任的UC与受信任的操作系统隔离开来。SEUSS[1]的方法与其他硬件实施的保护机制兼容，例如在虚拟机中隔离单内核。

通过将不受信任的单内核和主机之间的域接口缩小到一组受限制的系统调用来实现额外的保护。狭窄的域接口限制了受感染的客户端可用于发起攻击的外围应用（即，大多数内核功能在隔离域中处理），并使系统能够更轻松地监视和检测恶意行为。例如，我们的原型ukvm 中使用的超级调用接口仅公开12 个系统调用，而Docker 容器的标准安全性允许访问300 多个Linux 系统调用。

SEUSS[1]使用快照，通过将共享限制为函数历史时间线内的只读页面来保持用户之间的隔离。在将任何特定于函数的信息导入单内核之前捕获运行时快照，从而允许不同用户的函数共享相同的基本快照。从这些快照部署的UC 对快照中的一组共享只读页面具有写入时复制访问权限。因此，所有写入都会捕获到专门用于发布它们的UC 的新页面上。

页面共享的普遍使用确实意味着我们的应用程序容易受到侧信道攻击和硬件级漏洞的影响。例如，Rowhammer漏洞可用于破坏快照中的内存，该快照在数千个函数之间共享。我们的方法与现有的Rowhammer防御兼容。与KSM 相比，SEUSS 中的页面共享不追溯应用，从而减少了对基于重复数据删除的侧信道攻击的担忧。但是，处理内存读取的时间可以用作侧信道，以泄漏有关共同运行活动的信息。适用的软件技术已被证明可以减轻基于内存的侧信道攻击。

6 未来工作方向

SEUSS[1] 利用计算缓存来实现快速函数启动、更高密度的函数缓存，并支持FaaS 平台上的突发请求。未来的FaaS平台将继续需要这些属性，但其规模和并行程度远远超过单个计算节点。我们认为SEUSS 的自然演变是跨越节点的，以提供分布式和复制的全局缓存。单内核快照的只读和随处部署特性表明，它们可以在具有类似硬件配置文件的机器上克隆和部署。分布式SEUSS 将使高级共享技术能够加速远程部署，例如虚拟机状态着色或按需分页。

计算缓存框架允许我们研究更积极的预期优化方法。例如，我们正在探索使用连续硬件跟踪和机器学习来自动识别快照中的优化机会。

对于未来的工作，我们可以将MPSC 集成为Serverless 框架的插件，以通过统一的接口增加支持的提供商。此外，需要将Amazon S3 和IBM Cloud Object Storage 等存储提供商添加到MPSC 中。MPSC 框架的这些增强功能将大大提高其在调度无服务器应用程序方面的价值和多功能性。

7 结论

我们详细探讨了无服务器计算的起源和历史。它是云编程模型中抽象化趋势的演变，目前以功能即服务（FaaS）模型为例，在该模型中，开发人员编写小型无状态代码片段，并允许平台以容错方式管理可扩展执行功能的复杂性。

然而，这种看似限制性的模型非常适合许多常见的分布式应用程序模式，包括计算密集型事件处理管道。大多数大型云计算供应商都发布了自己的无服务器平台，并且围绕这一领域的行业进行了大量的投资和投入。

在这个领域存在各种各样的技术难题和智力上的问题，从基础设施问题，如优化到冷启动问题，再到可组合编程模型的设计。甚至还有哲学问题，例如分布式应用程序中状态的基本性质。许多开放问题是当今无服务器计算从业者面临的实际问题，解决方案具有潜在的重大影响。

无服务器范式最终可能会带来新的编程模式、语言和平台架构，这无疑是研究界参与和贡献的一个令人兴奋的领域。

无服务器技术简化了微服务开发，同时消除了服务器管理任务。因此，无服务器应用程序的开发人员几乎无法控制其代码的性能。为了解决这个问题，[3]提出的MPSC 框架在FaaS 提供商的生态系统中提供了卓越的用户定义的QoS 指标。结果显示，与在任何一个云提供商上执行相比，往返执行时间缩短了200%，并且在使用受限边缘计算资源时提高了系统敏捷性。因此，可以得出结论，利用包括边缘部署在内的多个FaaS 提供商可以显着提高无服务器应用程序的QoS。MPSC 框架的这些增强功能将大大提高其在调度无服务器应用程序方面的价值和多功能性。

参考文献

[1] James Cadden et al. “SEUSS: skip redundant paths to make serverless fast”. In: (2020), pp. 1–15.

[2] Scott Hendrickson et al. “Serverless Computation with OpenLambda.” In: 4th ser. 41 (2016).

[3] Austin Aske and Xinghui Zhao. “Supporting Multi-Provider Serverless Computing On The Edge”. In: (2018), 20:1–20:6.

[4] Paul C. Castro et al. “Serverless Programming (Function As A Service)”. In: (2017), pp. 2658–2659.

[5] Ioana Baldini et al. “Serverless Computing: Current Trends and Open Problems.” In: abs/1706.03178 (2017).