使用 Docker 和 Kubernetes 对多语言微服务应用进行容器化

Vamsi Krishna Yepuri、Venkata Kalyan Polamarasetty、Shivani Donthi、Ajay Kumar Reddy Gondi 计算机科学系，肯特州立大学

arXiv:2305.00600v1 [cs.SE] 30 Apr 2023

摘要——本项目探讨了容器化技术在现代软件开发和部署中的优势。研究重点介绍了在开发过程中使用 Kubernetes 和 Docker 的优点，包括微服务的便捷打包与部署、高效的资源利用率、更快的启动时间，以及更强的可扩展性和灵活性。本项目最后提出一项研究计划：使用 Java、Python 和 JavaScript 创建多语言微服务应用，使用 Docker 将其容器化，并部署到 Kubernetes 中。该研究旨在评估分布式模式下的服务发现和自动扩缩容，并比较虚拟机与容器的性能指标。研究结果可帮助软件开发团队了解容器化在现代软件开发和部署中的优势。

索引词——**Docker、Kubernetes、容器、微服务、多语言**

一、引言

容器化借助 Docker 改变了软件开发和部署方式。Docker 是应用最广泛的开源平台之一，可用于构建、交付和运行分布式应用程序。然而，随着容器化应用程序变得日益复杂，大规模管理它们也愈发具有挑战性。Kubernetes 是一种开源容器编排系统，为大规模管理和部署容器化应用程序提供了平台。它之所以广受欢迎，得益于其能够自动化部署、扩缩容和容器化应用程序管理，并提供自动负载均衡、扩缩容和自愈等功能。Kubernetes 技术的发展趋势包括采用无服务器计算，使开发人员无需管理基础设施或容器编排，即可在轻量级容器中运行代码。Kubernetes 还应用于混合计算和边缘计算场景，在这些场景中，计算资源分布于多个设备和地点，展现出其适应现代软件开发与部署需求的多功能性和适应能力。

公有云服务提供商在容器技术的普及和发展中发挥了重要作用。它们提供容器服务，使开发人员能够部署、管理和扩缩容容器化应用程序，从而降低了采用容器的门槛，让各种规模的企业都能使用容器。这些提供商还积极推动容器标准并促进互操作性，助力 Kubernetes 成为容器编排领域事实上的标准。容器编排标准化简化了在不同云服务提供商和本地部署环境中部署和管理容器化应用程序的过程。

容器在微服务架构中带来诸多好处，其中最显著的一点是能够轻松打包和部署微服务。每个微服务都可以打包为容器镜像，从而将其与底层基础设施解耦，使服务能够在各种环境中更轻松地部署和运行，同时避免兼容性问题。例如，考虑一个基于微服务的电子商务应用程序，其中包含产品目录、购物车、支付处理和订单管理等各种服务。每项服务都可以进行容器化，从而轻松实现独立部署和更新。这使开发人员能够添加新功能或修改现有功能，而不会影响其他服务。

此外，容器在微服务之间提供了高度的隔离，从而最大限度地减少它们之间的冲突和依赖关系。通过将每个服务打包为容器，每个服务都可以运行所需数据库的独立实例，而不会影响系统中的其他服务。因此，容器在微服务项目中提供了更高的敏捷性、可扩展性和一致性，使开发人员能够专注于开发和更新微服务，而不是基础设施和部署流程。

在运行应用程序时，使用虚拟机（VM）可能会带来挑战，因为虚拟机需要各自的操作系统，并且会消耗大量 CPU、内存和存储资源，这可能导致应用程序性能变慢并增加复杂性。通过使用容器技术可以解决这一问题，因为容器技术允许多个容器共享同一个操作系统内核，从而实现更高的资源利用率和更快的应用程序启动速度，并最终带来更高的可扩展性、灵活性和成本效益。

例如，考虑一个需要快速、高效地处理大量数据的数据分析应用程序。如果该应用程序运行在虚拟机上，由于运行完整操作系统所产生的开销和复杂性，可能会遭遇较慢的性能。然而，通过使用容器技术，开发人员可以在同一个宿主操作系统上运行多个轻量级容器，从而为应用程序带来更快的性能、更低的资源使用量和更高的可扩展性，并实现更快速的数据处理和分析。

由于容器与虚拟机这两种技术存在根本差异，因此使用虚拟机实现容器的能力颇具挑战。容器轻量且高效，与主机共享同一个操作系统内核；而虚拟机则会模拟整个硬件环境，包括独立的操作系统，这可能导致资源使用量更高、复杂性更大。此外，虚拟机缺乏可移植性和灵活性，因为每台虚拟机都需要特定的硬件和操作系统配置，因此难以在不同环境之间迁移或复制。另一方面，容器可以轻松打包，并部署到任何支持相同容器运行时的宿主系统上。尽管虚拟

机有自身的优势和适用场景，但由于效率更高、灵活性更强、可移植性更好，而且启动速度快得多，因此容器更适合大多数现代应用程序，也更适用于现代应用程序部署和扩缩容。

传统的单体应用程序方法需要构建一个功能全面的单一应用程序，开发和维护可能耗时费力，因此难以实现敏捷性。为了解决这些问题，并提升灵活性、简化维护，业界正转向微服务。在微服务时代，许多独立模块被部署在传统虚拟机中，这可能导致库之间发生冲突，也使资源配置、扩缩容、服务发现、负载均衡和部署都需要手动完成，既耗时又费力。为解决这些问题，可以将微服务打包到容器中，而非虚拟机中，从而有助于消除冲突，并使各个微服务彼此独立。借助 Kubernetes，可以以可扩展的分布式方式运行工作负载，从而解决自动扩缩容和服务发现方面的问题。本项目将从零开始，使用 Java、Python 和 JavaScript 构建一个多语言微服务 [1]应用程序，使用 Docker 将其容器化，并部署到 Kubernetes 中，以考察分布式模式下的服务发现和自动扩缩容，同时比较容器化技术与传统虚拟机方案的性能差异。

在本项目中，我们旨在通过多项指标比较虚拟机和容器的性能。首先，我们将考察这两种技术在微服务应用程序中的资源利用率，分析 CPU、内存和存储使用情况，以确定哪种技术效率更高。其次，我们将比较虚拟机和容器的启动时间及扩缩容能力，看看哪种技术速度更快、可扩展性更强。第三，我们将考察虚拟机和容器各自占用的存储空间，以了解哪种技术对存储的利用更有效。最后，我们将评估虚拟机和容器的可移植性，以及它们在不同环境之间迁移的能力。通过比较这些性能指标，我们旨在确定哪种技术更适合我们的微服务应用程序，并深入了解每种技术的优势和不足。

II. 相关工作

微服务、Docker、Kubernetes 和多语言技术是软件开发领域中共同使用的一些主要技术。关于这些主题有许多研究，阐述了转向 Docker 和 Kubernetes 的目的，以及这些方法所涉及的挑战。 [2]解释了采用基于微服务的架构来实现 Kubernetes 中有状态应用的高可用性的重要性。该研究探讨了在 Kubernetes 中管理有状态微服务的挑战，并提出了使用状态控制器来提高其可用性的解决方案，但未提及无状态多语言微服务的性能。[3],介绍了一个 Web 应用程序的案例研究，该应用程序由多个微服务组成，每个微服务都运行在独立的容器中，并强调了其可移植性、可扩展性和灵活性的提升。在某些情况下，为了实现预期的解决方案，需要使用多语言技术。 [4] 讨论了如何在使用多个数据库的多语言微服务之间实现一致的分布式事务，以及解决了多语言环境中分布式事务面临的挑战。[5]提出了一种性能模型，用于分析 Kubernetes 集群中的资源管理。该模型使用排队论和随机过程来表示节点和 Pod 的资源利用率。研究通过实验验证了所提出模型的有效性，并将其准确性与实际系统性能进行了比较。论文强调，拥有准确的性能模型对于优化资源分配、提高 Kubernetes 集群的整体效率至关重要。此外， [6]还评估了使用容器（尤其是 Docker）的微服务架构性能。作者开展实验，对比部署在 Docker 上的单体应用与基于微服务的应用的性能。实验测量了这些应用在各种工作负载下的响应时间和吞吐量。实验结果表明，在响应时间和可扩展性方面，微服务架构优于单体架构。总体而言，近年来，微服务、Docker、Kubernetes 和多语言环境在软件开发中的应用日益普及，这得益于它们在可移植性、可扩展性和灵活性方面的优势。不过，使用这些技术也面临一些挑战，例如在 Kubernetes 中管理有状态微服务，以及确保跨多语言微服务的分布式事务保持一致。为应对这些挑战，研究人员提出了各种解决方案和方法，例如使用状态控制器提高 Kubernetes 中有状态微服务的可用性，以及开发性能模型来优化 Kubernetes 集群中的资源分配。此外，研究人员还开展了使用 Docker 等容器评估微服务架构性能的研究，结果表明，在响应时间和可扩展性方面，微服务架构可以胜过单体架构。总体而言，随着越来越多的组织寻求利用这些技术带来的优势，同时应对其应用过程中的挑战，微服务、Docker、Kubernetes 和多语言环境在软件开发中的应用很可能会持续增长。

从表 *I*：相关工作中，我们可以看到参考论文实现的不同功能，以及本项目实现的功能。我们提出的解决方案将处理一个无状态多语言微服务，使用 Docker 将其容器化，并采用 Kubernetes。我们部署应用程序，并使用 Kubernetes 分析性能提升。

参考	微服务	无状态	容器	Kubernetes	多语言的
[2]	✓	X	✓	✓	X
[3]	X	✓	✓	X	X
[4]	X	X	✓	X	X
[5]	✓	✓	✓	✓	X
[6]	✓	X	✓	X	✓
我们的项目	✓	✓	✓	✓	✓

表 I：相关工作

III. 项目背景

近年来，微服务架构日益流行，因为它使组织能够以更加敏捷、高效的方式开发、部署和管理大型复杂应用程序。然而，实施微服务架构面临的一项挑战是如何管理

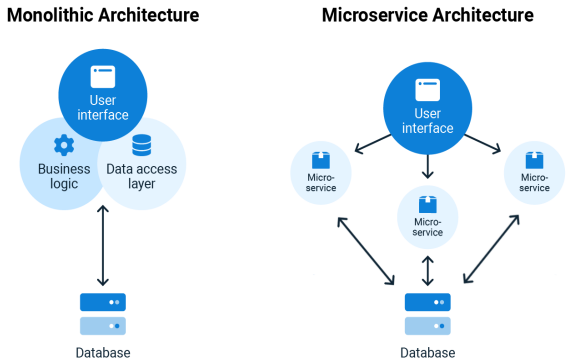

图 1：单体架构与微服务



图 2：多语言微服务

独立服务的部署和扩缩容，同时兼顾效果与效率。这正是 Docker 和 Kubernetes 等容器化及容器编排工具发挥作用的地方。在本项目中，我们将展示使用容器编排工具带来的性能提升分析。

A. 微服务与单体架构

单体架构和微服务架构是设计和构建软件系统的两种不同方法。单体架构是一种传统方法，即将应用程序构建为一个整体；而微服务架构则是将应用程序拆分为多个较小且相互独立的服务。图 *1*：单体架构与微服务展示了单体架构与基于微服务的架构之间的区别。单体方法更容易开发和部署，但随着应用程序的增长，维护和扩展可能会变得困难。另一方面，微服务方法具有更大的灵活性、可扩展性和韧性，但需要不同团队之间进行协调。架构的选择取决于应用程序的具体需求和要求：单体架构更适合较小的应用程序，而微服务则适用于较大且复杂的应用程序。

[7]介绍了将单体应用转换为微服务的过程，并提供了一个用于理解迁移过程的框架。文中讨论了采用微服务架构的优势与挑战。

B. 多语言微服务

多语言微服务是指一种软件架构：使用多种编程语言和技术构建相互协作、共同实现更大目标的独立小型服务。图 *2*：多语言微服务展示了多语言微服务。这种方法让开发人员能够为每项服务选择最合适的语言和工具，而不必局限于单一技术栈。多语言微服务日益普及，因为它们具备多项优势，例如更高的灵活性、更强的可扩展性和更好的容错能力。它们可以通过 API、消息队列或其他协议相互通信，从而作为一个协调一致的系统协同工作。构建和管理多语言微服务也面临一些挑战，例如需要管理多种编程语言，以及相应的工具，但采用恰当的方法和工具便能克服这些挑战。因此，许多组织正在采用这种方法，构建更具韧性和适应性的软件系统。

多语言微服务架构在监控 [8]，负载均衡 [9]，安全 [10]和服务质量（QoS） [11] 方面具有显著优势，因为它允许您为每个微服务选择最合适的技术、增强韧性、简化监控和故障排查、分配工作负载，并针对特定任务或功能优化每个微服务。多语言微服务架构还可以利用数据中心网络中的流量工程！ [12]能力，优化微服务性能 [13] 并提升整体应用程序性能。通过采用智能路由、流量整形、负载均衡、网络性能监控和可扩展性，多语言微服务架构可以确保流量路由至最优端点，管理微服务之间的流量，均衡分配流量至多个端点，监控网络性能，并适应流量规模和应用程序使用情况的变化。所有这些能力都有助于降低延迟、防止拥塞、高效分配资源、快速识别和解决问题，并最终提升应用程序性能和用户体验。

C. 容器

容器是一种轻量、独立且可执行的软件包，其中包含运行应用程序所需的一切内容，例如代码、库和系统工具。容器提供一致的运行时环境，确保应用程序无论底层系统配置如何，都能可靠运行。图 *3*：虚拟机与容器展示了虚拟机与容器之间的区别。[14]是一份关于容器技术在现代软件开发中应用情况的综合报告，涵盖容器技术的历史、优势、挑战和未来趋势等多个方面。容器化的目标是将每项服务及其依赖关系打包到一个轻量、可移植的容器中，使其能够在任何机器或云环境中运行。

Docker 被广泛用作容器化平台，因为它为运行应用程序提供了可靠且可复现的环境。Docker 让创建、部署和管理镜像变得简单，从而更便于在不同环境中开发和测试应用程序

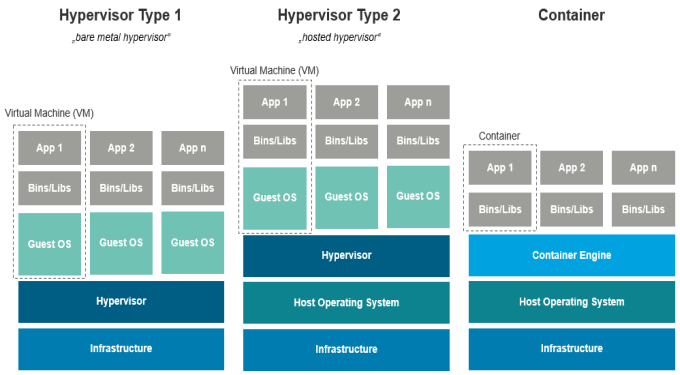

图 3：虚拟机与容器

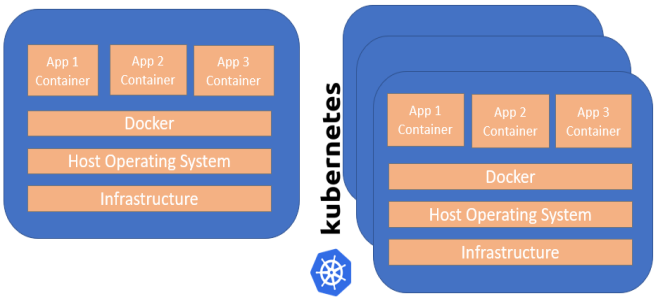

图 4：Docker 与 Kubernetes

。Docker 容器是一个独立运行的软件包，不受其他容器和宿主系统的影响。它拥有自己的网络、文件系统和资源，非常适合部署具有特定需求的应用程序。Docker 镜像是应用程序文件系统的快照，可以通过 Dockerfile 脚本创建。镜像可用于启动一个或多个容器。Docker 容器的优点包括可移植性、一致性、隔离、资源效率和可扩展性；缺点则包括复杂性、学习曲线、安全问题、依赖项管理方面的挑战，以及对硬件资源的访问受限。

D. Kubernetes 和 Docker

Kubernetes 是一个用于容器编排的开源平台，可自动执行容器化应用程序的部署、扩缩容和管理。图 *4*: *Docker 与 Kubernetes*展示了 Docker 与 Kubernetes 之间的区别，以及 Kubernetes 如何在分布式环境中使用并编排 Docker 容器。它提供了一个框架，可在服务器集群中部署、扩缩容和管理多个容器，使组织能够大规模运行和管理其微服务。[15] 说明了如何使用 Docker 和 Kubernetes 自动扩缩云环境中的国防应用程序。它重点介绍了这些工具的优势，例如提高安全性和可靠性，并提供了详细的部署流程。

对于采用多种编程语言和框架开发独立服务的多语言微服务应用程序而言，容器化以及容器编排尤其有价值。Docker 和 Kubernetes 使组织能够轻松打包和部署以不同语言或框架编写的服务，同时确保每个服务都在一致且隔离的环境中运行。这让管理依赖关系、测试和部署服务，以及根据需要扩展应用程序变得更加容易。

总体而言，使用 Docker 和 Kubernetes 对多语言微服务应用程序进行容器化，可以帮助组织在软件开发和部署流程中实现更高的敏捷性、可扩展性和可靠性。这也有助于降低管理和扩缩容多个服务的复杂性，而这在多语言环境中尤其具有挑战性。

IV. 基础设施

使用 Docker 和 Kubernetestypically 进行容器化所需的基础设施通常包括：

a. 物理机或虚拟机，b. 存储系统，以及 c. 网络组件。

需要物理机或虚拟机来托管容器化应用程序。这些机器应具备足够的 CPU、内存和存储空间，以运行应用程序和容器。存储系统用于存储容器镜像，以及应用程序需要访问的任何数据或文件。基础设施中的所有机器都应能够访问这些存储。还需要负载均衡器 [9] 和防火墙等网络组件，以确保流量能够路由到相应的容器，并保障网络安全。我们还需要云引擎，因此使用 GCP。谷歌云平台 (GCP) 是一家云服务提供商，提供一系列服务，用于在云中托管和管理应用程序、数据和基础设施。GCP 提供的工具和服务包括 Google Kubernetes Engine (GKE)、GoogleContainer Registry (GCR) 和 Cloud Storage，非常适合使用 Docker 和 Kubernetes 进行容器化。此外，GCP 还提供用于监控和管理容器化应用程序的工具，例如 Stackdriver Monitoring and Logging，以及用于应用程序跟踪的 Cloud Trace。

在 [16] 中提供了使用谷歌云平台部署 Web 应用程序的示例，其中包括创建新项目、启用必要的 API、选择部署方法、创建实例或应用程序、进行配置以及部署应用程序等步骤。部署完成后，可以通过谷歌云平台对该应用程序进行管理和监控。

此外，所使用的 GCP 服务包括虚拟私有云网络（ Virtual Pri- vate CloudNetworks）和计算引擎。虚拟私有云网络支持在 GCP 中创建私有虚拟网络，而计算引擎提供具有可配置 CPU、内存和存储容量的虚拟机，用于运行 Kubernetes 以及部署应用程序和服务。

V. 系统架构

为了在我们的场景中使用应用程序测试 Docker 和 Kubernetes，需要一个采用多种编程语言设计的多语言微服务应用程序。书店应用由此登场。书店应用作为一个多语言微服务，旨在为顾客提供便捷的在线浏览和购买图书的方式。如图 5 所示，该应用由四个微服务共同实现，这些服务使用多种编程语言和脚本开发。它们分别是

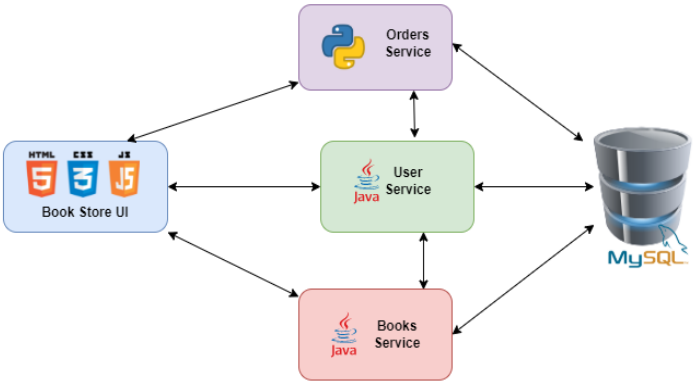

图 5：书店应用架构

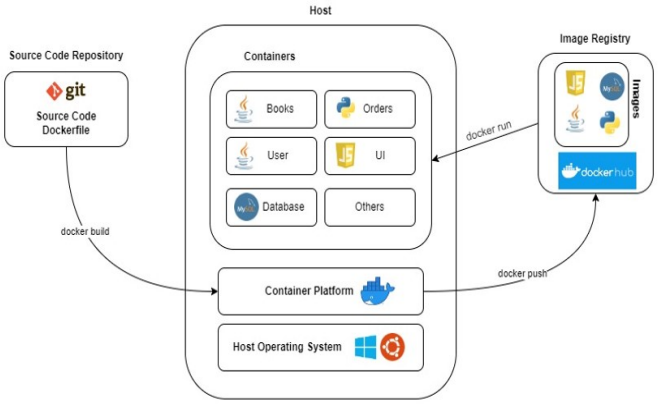

图 6：单节点 Docker 系统上的书店应用架构

用户服务、订单服务、图书服务和书店用户界面服务。数据存储在数据库的中央存储库中。所有微服务彼此相连，并通过 REST API 进行通信。假设书店应用部署在如图 6 所示的单节点 Docker 系统上。Docker 容器是轻量级、独立的虚拟机，可在任何计算机上运行。Docker 引擎可有效管理这些容器，确保它们能够访问内存和 CPU 等必要资源，同时彼此隔离。通过在同一台机器上运行多个容器，并为每个容器配置各自的依赖项和配置，应用程序便可轻松部署和管理，无需担心不同软件组件之间发生冲突。所有服务及其源代码和 Dockerfile 都通过 Docker 容器平台构建。构建完成后，数据会被推送到 Docker Hub 的中央存储库。这样便无需为每个微服务单独下载和配置软件依赖项，从而提高部署效率并减少耗时。由于所需软件已预先打包到镜像中，只需拉取所需镜像并在 Docker 容器中运行即可。要运行书店应用，只需使用 docker run 命令启动存储在 Docker Hub 上的所有必需微服务镜像。执行 docker 给出了使用保存在 Docker Hub 中的镜像名称运行的命令。从单节点 Docker 系统迁移到 Kubernetes 的主要原因是，Kubernetes 能为用户需求更高、更加复杂的应用程序带来多项优势，例如可扩展性、高可用性、资源效率和服务发现。通过将工作负载分配到多个节点上，Kubernetes 能够处理远超单节点 Docker 系统的工作负载，因此非常适合流量快速增长的应用程序。Kubernetes 的自动检测和恢复功能有助于保持应用程序的可用性、减少停机时间，并尽可能减少人工干预。

书店应用可以部署在 Kubernetes 上。对于多节点部署，我们使用谷歌云平台 GCP，如图 7 所示。系统首先创建一个 VPC 网络，它是用于谷歌云资源的虚拟专用网络。该网络为 Kubernetes 集群的部署提供了一个专用网络空间。随后，在 VPC 网络上部署一个 Kubernetes 集群，其中包含 1 个主节点和 2 个从节点。该集群由一个或多个节点组成，每个节点都是一台虚拟机（VM）。每台虚拟机运行一个 Kubernetes 节点，负责运行 Pod 并管理虚拟机的资源。利用 GCP 中的计算引擎部署一个 Kubernetes 主节点，由它管理 Kubernetes API 服务器、控制器、调度器和 etcd 数据库。API 服务器负责接收并处理来自客户端的请求，而 etcd 则存储 Kubernetes 集群的配置数据和元数据。控制器向从节点发送命令。例如，如果要将镜像部署到从节点，控制器会向该从节点上的 kubelet 发送命令，以执行此过程。在从节点上初始创建 Pod 时，调度器用于配置 Pod 的放置位置，以应对可能发生的停机。容器启动时，由容器运行时负责运行容器。我们将使用网络文件系统（NFS）服务器，将数据库中存储的数据保存在一个中央存储库中。随后，Kubernetes 客户端可以与 API 服务器交互，在集群上部署和管理应用程序。部署应用程序时，Kubernetes 会根据资源需求和可用性，将所需的容器调度到可用节点上，并管理其生命周期，按需扩容或缩容。用户通过部署在 Kubernetes 上、作为服务提供的负载均衡器访问应用程序。该服务会将流量分配到 Kubernetes 集群中运行的容器。

VI. 实现

本项目按以下四个步骤实现。为了对多语言微服务应用进行容器化，我们首先需要开发应用程序所需的微服务，然后将这些微服务的镜像部署到已配置 Kubernetes 的云系统中。部署微服务后，我们将比较应用程序在虚拟机和容器中的性能。

A. 多语言微服务的开发及使用 *Docker* 将其容器化

开发了一款书店应用，用于在线订购图书。顾客可以浏览图书、将图书加入购物车、下单，并查看订单历史和订单状态。

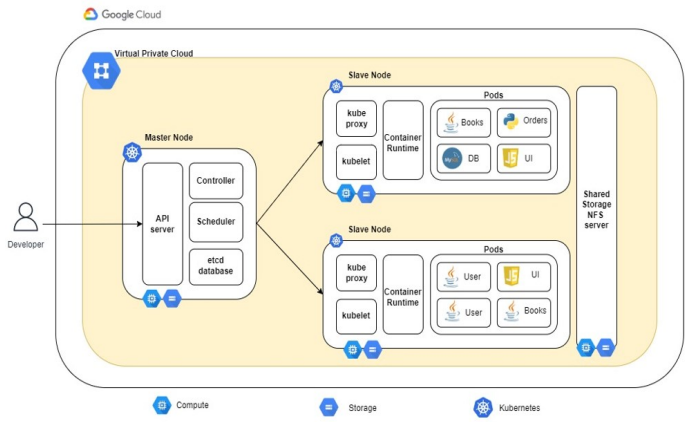

图 7：Kubernetes 上书店应用的架构

管理员也可以轻松地向应用程序中添加新图书，包括作者信息、价格和库存数量等重要详情。为实现这些功能，书店应用由四个关键微服务组成：用户服务、订单服务、前端书店用户界面和图书服务。这些微服务协同工作，确保应用程序发挥最佳性能。数据存储在一个集中式数据库 MySQL 中，该数据库因其可靠性和性能而经过精心选用。书店用户界面使用 HTML、CSS 和 JavaScript 开发，为顾客和管理员提供了友好直观的界面。用户服务会根据用户角色，确保其能够使用相应的功能。该服务使用 Java 编程语言开发。订单服务负责管理用户的购物车详情、订单历史以及每笔订单的配送状态。该服务使用 Python 编程语言开发，并与其他微服务无缝集成。图书服务使用 Java 编程语言开发，负责管理应用程序中可供购买的图书信息。开发这款应用程序时，我们十分注重细节，确保从浏览图书到订单履行的整个流程顺畅直观，并以多语言微服务的形式开发。应用程序开发完成后，我们使用 Dockerfile 将其构建为 Docker 镜像，并将这些镜像推送到 Docker Hub。接着，我们设置了一台单节点 Docker 机器，并从 Docker Hub 将所有 Docker 镜像拉取到这台机器上。最后，我们在这台单节点 Docker 机器上运行所有容器，从而使用 Docker 测试应用程序。

B. 在云端设置 *Kubernetes*

为了大规模部署书店应用，Kubernetes 设置在谷歌云平台（GCP）上完成。谷歌云平台会自动配置虚拟私有云（VPC）网络，用于应用程序之间的相互通信和网络发现。作为项目的一部分，我们创建了一个包含三个节点的 Kubernetes 集群，配置为 4 个 vCPU 和 8GB 随机存取存储器。我们正在实施的集群包含一个主节点和两个从节点。机器上部署的操作系统是基于 2023-03-02 构建的 Ubuntu 20.04 LTS amd64 focal 映像。我们使用 containerd 作为容器运行时。

我们另外创建了一个节点，配置为 4 个 vCPU、8GB 随机存取存储器和 50GB 固态硬盘，操作系统为基于 2023-03-02 构建的 Ubuntu 20.04 LTS amd64 focal 映像。该机器用作网络文件系统 - NFS 服务器，以提供集中式存储。在 Kubernetes 环境中，NFS 服务器可用作持久化存储解决方案，使容器能够访问和操作存储在网络文件系统上的数据。Kubernetes 集群配置为自动预配和管理由 NFS 服务器提供支持的持久卷（PVs），供运行在 Pod 中的应用程序使用。当 Pod 请求访问存储时，它会创建一个持久卷声明（PVC），其中指定存储需求；Kubernetes 将使用该声明动态预配由 NFS 服务器提供支持的持久卷（PV）。持久卷预配完成后，可以将其作为卷挂载到 Pod 的容器中，使应用程序能够读取和写入共享存储中的数据。这为存储数据提供了一种可扩展且可靠的方式，使多个 Pod 或节点能够在 Kubernetes 集群中访问这些数据。

C. 在 *K8s* 上进行微服务部署、扩缩容和负载均衡

在这一阶段，我们创建了部署书店应用所需的清单文件。在 Kubernetes 中，部署文件和服务文件是两种用于定义和管理集群中运行的应用程序的清单文件。对于每个微服务，都会创建一个部署文件，并在 Kubernetes 中将其作为部署对象；该对象继而在 Pod 内运行应用程序容器。部署文件通常包含容器镜像、要创建的副本数量，以及必要的环境变量、卷和网络配置等规范。将部署文件应用到集群后，Kubernetes 会创建指定数量的副本并管理其生命周期，确保始终有预期数量的副本运行且保持最新。这负责对各个微服务进行扩缩容，并自动修复 Pod。所有微服务的服务文件也已创建，因为它们用于定义 Kubernetes Service 对象，为访问集群中运行的一组 Pod 提供稳定的 IP 地址和 DNS 名称。服务文件通常包含服务类型、用于选择目标 Pod 的选择器，以及必要的网络配置等规范。将服务文件应用到集群后，Kubernetes 会创建 Service 对象，并确保集群中的其他对象可以使用它来访问这些 Pod。默认情况下，流量会通过轮询调度算法分发到各个 Pod。MySQL 数据库被配置为持久卷，因为存储在 MySQL 数据库中的数据会在 Pod 重启或节点故障后继续保留，从而确保数据始终可用。此外，还会使用集中式存储解决方案（例如 NFS 服务器或基于云的存储服务）来存储数据。这有助于确保数据安全、集中存储，并便于 Kubernetes 集群中的其他 Pod 或服务访问。我们确保书店应用的全部四个微服务都作为 Pod 在 Kubernetes 集群中运行，并且能够彼此通信，也能够与数据库 Pod 通信。接下来，我们通过创建服务将应用程序暴露到互联网。这样，用户便可使用集群中任意节点的 IP 地址访问应用程序。完成这些步骤后，书店应用已成功启动并运行，用户可以通过集群中任意节点的 IP 地址经由互联网访问它。

D. 虚拟机和容器上应用程序的性能比较

评估容器与虚拟机的性能，是为特定应用程序或工作负载确定最合适的虚拟化技术的重要步骤。在此评估中，考虑的关键指标包括

1. 启动时间 2. 服务器
占用空间 3. 性能指标

为了评估容器化服务和运行在虚拟机上的服务的性能，我们在相同的环境中测试两种系统，并为其分配相同的资源：2 核 CPU 和 2 GB 随机存取存储器，以获得尽可能好的结果。

1) 启动时间：启动时间是指服务器或系统启动并完全投入运行所需的时间。这是一个需要考虑的重要参数，尤其是在高可用性和恢复时间至关重要的情况下，因为启动时间过长会导致停机时间延长，并可能影响用户体验。影响启动时间的因素包括硬件配置、软件栈和网络连接。在我们的场景中，应用程序作为服务运行在虚拟机和容器中，这意味着应用程序将在系统启动后启动。这里我们考察虚拟机和容器各自的启动时间。

a. 在虚拟机上运行的应用程序的启动时间：为了计算服务器和应用程序启动所需的启动时间，需要计算虚拟机开机时间戳与应用程序开始提供服务的时间戳之间的差值。

b. 容器中运行的应用程序的启动时间：要计算容器中运行的应用程序的启动时间，需要计算现有 Pod 被终止的时间戳与下一个 Pod 开始提供服务的时间戳之间的差值。

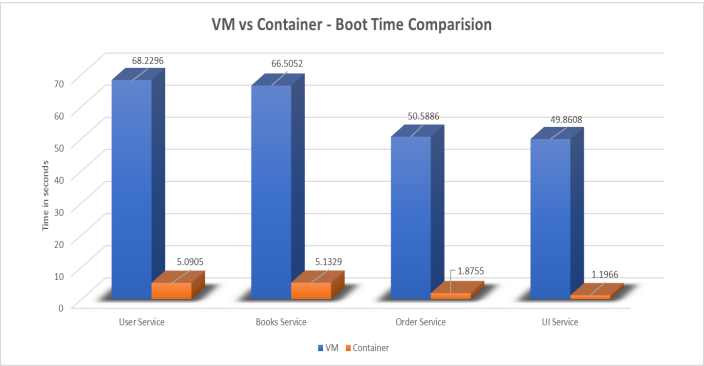

图 8：虚拟机与容器对比 - 启动时间比较

图 *8*：虚拟机与容器——启动时间对比 展示了虚拟机和容器的启动时间对比。显然，使用容器化服务后，启动时间得到了大幅缩短。

2) 服务器占用空间：服务器占用空间是指服务器所占用的物理空间大小。在空间有限的情况下，较小的服务器占用空间可能更为理想。

图 *9*：虚拟机与容器——服务器占用空间对比展示了与虚拟机相比，容器大幅减少了服务器占用空间。

3) 性能指标：性能指标是指用于衡量服务器或系统性能的各项指标，例如 CPU 利用率、内存使用率和响应时间。为了确保

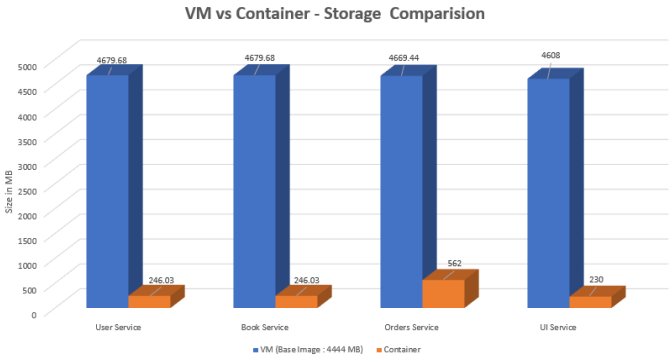

图 9：虚拟机与容器——服务器占用空间对比

系统能够满足用户需求并以最佳状态运行，需要考虑这些指标。影响性能指标的因素包括硬件配置、软件栈、网络连接和工作负载特征。我们测试了虚拟机和容器的性能指标，并观察了两种状态下的性能指标。一种是空闲状态，此时服务器未处理任何流量；另一种是负载状态，此时我们模拟应用程序上的负载。在这两种不同状态下，我们观察了容器和虚拟机的性能指标。

在负载状态下，即为了模拟应用程序的负载，我们使用 JMeter 工具模拟负载，在 1 秒的渐增时间内发送 1000 个请求，并进行 10 次迭代。因此，总共将向应用程序施加 10000 个请求。使用 Prometheus 监控工具对虚拟机和容器在空闲状态和负载状态下的情况进行观测。为了可视化这些指标，我们在 Prometheus 之上使用 Grafana 创建仪表盘，以便观测性能。每个虚拟机和容器均分配了 2 核 CPU 和 2 GB 内存的资源限制。下文将比较我们的多语言微服务在虚拟机和容器两种环境下、不同状态（空闲状态和负载状态）中的性能指标。

a) 订单 - *Python* 服务：订单微服务使用 Python 编写，用于管理我们应用程序中的订单。



图 10：订单 Python 服务 - 空闲状态对比

图：*10* 订单 *Python* 服务 - 空闲状态对比 展示了应用程序在虚拟机和容器中处于空闲状态时的 CPU 利用率和内存消耗对比。处于空闲状态时，虚拟机的 CPU 使用率在 1.4% 到 2.1% 之间波动，而容器中的 CPU 使用率在 0.004% 到约 0.006% 之间波动。虚拟机中订单服务的内存使用率起初约为 41.36%，随后升至约 41.3725% 并保持稳定，而容器中起初约为



图 11：订单 Python 服务——负载状态对比

VM

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	780	13	15064	1718.21	0.00%	646.0/sec	253.60	84.53	402.0
TOTAL	10000	780	13	15064	1718.21	0.00%	646.0/sec	253.60	84.53	402.0

Container

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	625	7	15565	1544.61	0.00%	613.9/sec	241.01	80.94	402.0
TOTAL	10000	625	7	15565	1544.61	0.00%	613.9/sec	241.01	80.94	402.0

图 12：订单 Python 服务——响应时间对比

8.5%，并增加至11%，随后保持了一段时间不变。图：*11* 订单 *Python* 服务 - 负载状态比较 展示了应用程序在虚拟机和容器的负载状态下的 CPU 利用率和内存消耗对比。当对运行在虚拟机上的订单服务施加负载时，我们可以看到 CPU 使用率显著升高，一段时间内维持在约60%；而在容器中，我们同样可以看到明显的峰值，约为20%。虚拟机中订单服务的内存使用率起始约为41.4%，并增加至约41.6%；而在容器中，起始约为14%，增加至18%，并在一段时间内保持不变。通过对运行在虚拟机和容器中的订单服务在空闲状态和负载状态下的 CPU 使用率及内存使用率进行上述比较，我们可以明确地说，容器中的订单服务消耗的资源更少。图：*12* 订单 *Python* 服务 - 响应时间比较展示了订单服务运行在虚拟机和容器中时的响应时间对比。在虚拟机和容器中运行应用程序时，均发送了10000个请求以检查响应时间；虚拟机上的平均时间约为780毫秒，而在容器中运行时，平均响应时间为625毫秒。通过上述比较，我们可以清楚地看到，该服务在容器中的表现良好。

b) UI - JavaScript 服务：UI 由一个使用 HTML 和 JavaScript 编写的微服务处理，用于实现用户与应用程序的交互。图 *13*：*UI JS* 服务 - 空闲状态对比展示了应用程序在虚拟机和容器处于空闲状态时的 CPU 利用率和内存消耗对比。处于空闲状态时，虚拟机的 CPU 使用率在 2% 到约 4% 之间波动，而容器的 CPU 使用率在 0.001% 到约 0.002% 之间波动。虚拟机中 UI 服务的内存使用率起初约为 18%，随后升至约 20.5% 并保持稳定；而容器中起初约为 1%，之后保持稳定。图 *14*：*UI JS* 服务 - 负载状态对比展示了应用程序在红黑大战虚拟机和容器上的 CPU 利用率和内存消耗对比。



图 13：UI JavaScript 服务——空闲状态对比



图 14：UI JavaScript 服务——负载状态对比

VM

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	376	9	2105	354.40	0.00%	703.6/sec	1376.24	85.20	2003.0
TOTAL	10000	376	9	2105	354.40	0.00%	703.6/sec	1376.24	85.20	2003.0

Container

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	140	10	1175	175.15	0.00%	3347.8/sec	6417.78	385.79	1963.0
TOTAL	10000	140	10	1175	175.15	0.00%	3347.8/sec	6417.78	385.79	1963.0

图 15：UI JavaScript 服务——响应时间对比

虚拟机和容器在负载状态下。处于负载状态时，虚拟机的 CPU 使用率在 25% 左右波动，而容器的 CPU 使用率在 5% 左右波动。虚拟机中 UI 服务的内存使用率起初约为 22.53%，并在负载期间升至约 22.67%；而容器在负载期间约为 1.8%。通过上述两项比较，我们可以看出，无论是在空闲状态还是负载状态下，UI JS 服务在容器中消耗的资源都少于在虚拟机中消耗的资源。图：*15 UI JS* 服务 - 响应时间比较 展示了 UI JS 服务在虚拟机和容器中运行时的响应时间比较。在虚拟机和容器中运行应用程序时，我们均发送了 10000 个请求以检查响应时间。虚拟机中的平均响应时间约为 376ms，而容器中的平均响应时间为 140ms。通过上述比较，我们可以清楚地看到，UI JS 服务在容器中运行时性能更好。

c) 图书 - *Java* 服务：图书微服务是用 Java 编写的，用于管理应用程序中的图书。图：*15* 图书 *Java* 服务 - 空闲状态对比展示了应用程序在虚拟机和容器中处于空闲状态时的 CPU 利用率和内存消耗对比。



图 16：Books Java 服务 - 空闲状态比较



图 17：Books Java 服务 - 负载状态比较

VM

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	327	6	3037	346.71	0.00%	1696.9/sec	473.95	220.40	286.0
TOTAL	10000	327	6	3037	346.71	0.00%	1696.9/sec	473.95	220.40	286.0

Container

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	417	8	7233	892.66	0.00%	1234.9/sec	344.90	161.59	286.0
TOTAL	10000	417	8	7233	892.66	0.00%	1234.9/sec	344.90	161.59	286.0

图 18：Books Java 服务 - 响应时间比较

CPU 使用率：处于空闲状态时，虚拟机的 CPU 使用率在 2.2% 到 2.8% 之间波动，而容器中的 CPU 使用率在 0.03% 到约 0.07% 之间波动。订单服务在虚拟机中的内存使用率约为 33%，而在容器中初始约为 12.77%。图：*16* 图书 *Java* 服务 - 负载状态对比展示了应用程序在负载状态下运行于虚拟机和容器时的 CPU 利用率和内存消耗对比。负载状态下，虚拟机的 CPU 使用率约为 50%，可以看到其数值在 0 到约 50% 之间出现峰值；而容器中的 CPU 使用率约为 17.5%。UI 服务在虚拟机中的内存使用率初始约为 32%，负载期间升至约 40%；而在容器中，初始约为 13%，负载期间升至 17%。通过以上两项对比可以看出，无论在空闲状态还是负载状态下，Java 服务在容器中的运行表现都优于在虚拟机中的表现。图：*17* 图书 *Java* 服务 - 响应时间对比展示了 Java 服务在虚拟机和容器中运行时的响应时间对比。在虚拟机和容器中运行应用程序时，发送了 10000 个请求以检查响应时间，平均响应时间约为 327 毫秒；而在容器中运行时，平均响应时间为 417 毫秒。通过上述比较，我们实际上不能说 Java 服务在容器中运行得更高效，因为容器情况下的响应时间更长。

VII. 结论

通过此次评估，在容器上运行的服务具有多项优势，包括更快的启动时间、更低的磁盘使用量，以及更低的 CPU 和内存使用率。它们为软件开发和部署提供了轻量且高效的解决方案，尤其适用于资源受限环境。然而，在响应时间至关重要且需要严格隔离的情况下，虚拟机更具优先性。容器作为主机上的一个进程运行，仅在应用层提供隔离。如果安全是主要考虑因素，与容器相比，虚拟机能够提供更高级别的隔离。

参考文献

[1] Pwint Phyu Khine 和 Zhaoshun Wang。大数据世界中的多语言持久化综述。信息，10(4):141，2019。[2] Leila Abdollahi Vayghan、Mohamed Aymen Saied、Maria Toeroe 和 Ferhat Khendek。基于微服务的架构：利用 Kubernetes 实现有状态应用程序的高可用性。载于 *2019 IEEE* 第 *19* 届软件质量、可靠性与安全国际会议（*QRS*），第 176–185 页。IEEE，2019。[3] Vivek Sharma、Harsh Kumar Saxena 和 Akhilesh Kumar Singh。面向多容器 Web 应用程序的 Docker。载于 *2020*年第二届工业应用创新机制国际会议（*ICIMIA*），第 589–592 页。IEEE，2020。[4] Guogen Zhang、Kun Ren、Jung-Sang Ahn 和 Sami Ben-Romdhane。Grit：使用多个数据库实现跨多语言微服务的一致性分布式事务。载于 *2019 IEEE* 第 *35* 届数据工程国际会议（*ICDE*），第 2024–2027 页。IEEE，2019。[5] Vıctor Medel、Omer Rana、Jos´e ´Angel Ba˜nares 和 Unai Arronategui。Kubernetes 中性能与资源管理的建模。载于 第九届公用事业与云计算国际会议论文集，第 257–262 页，2016。[6] Marcelo Amaral、Jorda Polo、David Carrera、Iqbal Mohomed、Merve Un- uvar 和 Malgorzata Steinder。使用容器对微服务架构进行性能评估。载于 *2015 IEEE* 第 *14* 届网络计算与应用国际研讨会，第 27–34 页。IEEE，2015。[7] Daniel Escobar、Diana C´ardenas、Rolando Amarillo、Eddie Castro、Kelly Garc´es、Carlos Parra 和 Rubby Casallas。探索单体应用程序向微服务演进的理解。载于 *2016* 年拉丁美洲计算会议（*CLEI*），第 1–11 页，2016。[8] Massimiliano Rak、Salvatore Venticinque、Gorka Echevarria、Gorka Esnal 等。云应用监控：Mosaic 方法。载于 *2011 IEEE* 第三届云计算技术与科学国际会议，第 758–763 页。IEEE，2011。[9] Debobroto Das Robin 和 Javed I. Khan。CLB：面向粗粒度精度的流量感知加权成本多路径负载均衡方法，应用于 PISA。*IEEE* 网络与服务管理汇刊，19(2):784–803，2022。[10] Uwe Zdun、Pierre-Jean Qu`eval、Georg Simhandl、Riccardo Scandariato、Somik Chakravarty、Marjan Jelic 和 Aleksandar Jovanovic。用于安全通信、身份管理和可观测性的微服务安全度量。*ACM* 软件工程与方法学汇刊，32(1):1–34，2023。[11] Debobroto Das Robin 和 Javed I. Khan。P4kp：使用可编程交换机实现 QoS 感知的前 k 条最优路径。*IEEE Access*，9:109115–109129，2021。[12] Debobroto Das Robin 和 Javed I Khan。P4te：基于 PISA 交换机的 Fat-tree 数据中心网络流量工程。计算机网络，215 109210，2022：。[13] Vittorio Cortellessa、Daniele Di Pompeo、Romina Eramo 和 Michele Tucci。面向微服务系统持续性能工程的模型驱动方法。系统与软件杂志，183 111084，2022：。[14] Tamanna Siddiqui、Shadab Alam Siddiqui 和 Najeeb Ahmad Khan。容器技术的综合分析。载于 *2019* 年第四届信息系统与计算机网络国际会议（*ISCON*），第 218–223 页，2019。[15] San Kho Lin、Umer Altaf、Glenn Jayaputera、Jiajie Li、Davis Marques、David Meggyesy、Sulman Sarwar、Shivank Sharma、William Voorsluys、Richard Sinnott、Ana Novak、Vivian Nguyen 和 Kristan Pash。使用 Docker 和 Kubernetes 跨云实现防御应用程序的自动扩缩容。载于*2018 IEEE/ACM* 公用事业与云计算国际会议伴随会议（*UCC Companion*），第 327–334 页，2018。

[16] Ambika Gupta、Pragati Goswami、Nishi Chaudhary 和 Rashi Bansal。使用谷歌云平台部署应用程序。载于 *2020*年第二届工业应用创新机制国际会议（*ICIMIA*），第 236–239 页，2020。