aws

最佳实践指南

# Amazon EKS



Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

# Amazon EKS: 最佳实践指南

Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。不属于亚马逊的所有其他商标均为其各自所有者的财产，他们可能与亚马逊有关联、关联或由亚马逊赞助，也可能不是

# Table of Contents

简介 ..... 1
相关指南 ..... 1
贡献 ..... 2
安全性 ..... 3
如何使用本指南 ..... 3
了解分担责任模型 ..... 3
简介 ..... 1
反馈 ..... 5
阅读更多内容 ..... 5
工具和资源 ..... 6
EKS 自动模式-安全 ..... 6
安全架构 ..... 7
常见问题 (FAQ) ..... 15
身份和访问管理 ..... 18
控制 EKS 集群的访问权限 ..... 18
集群访问建议 ..... 24
EKS 容器的身份和凭证 ..... 30
EKS pod 的身份和凭证推荐 ..... 37
工具和资源 ..... 45
Pod 安全 ..... 45
Linux 功能 ..... 45
Pod 安全解决方案 ..... 47
建议 ..... 54
工具和资源 ..... 45
Multi-tenancy ..... 60
软多租户 ..... 61
Kubernetes 构造 ..... 62
缓解控制措施 ..... 64
艰难的多租户 ..... 71
未来的方向 ..... 71
Multi-cluster 管理工具和资源 ..... 71
侦探控制 ..... 72
建议 ..... 54
工具和资源 ..... 45

网络安全 ..... 79
交通管制 ..... 80
网络加密 ..... 80
网络政策 ..... 80
建议 ..... 54
安全组 ..... 85
何时对 Pod 使用网络策略与安全组？ ..... 86
服务网格策略实施或 Kubernetes 网络策略 ..... 88
ThirdParty 网络策略引擎 ..... 88
传输中加密 ..... 90
工具和资源 ..... 45
数据加密和机密管理 ..... 98
静态加密 ..... 98
密钥管理 ..... 100
工具和资源 ..... 45
运行时系统安全性 ..... 103
安全上下文和内置 Kubernetes 控件 ..... 103
建议 ..... 54
工具和资源 ..... 105
基础设施安全性 ..... 106
建议 ..... 54
替代方案 ..... 109
工具和资源 ..... 45
监管合规 ..... 112
向左移动 ..... 114
工具和资源 ..... 45
事件响应和取证 ..... 115
事件响应计划示例 ..... 115
建议 ..... 54
工具和资源 ..... 45
映像安全 ..... 119
建议 ..... 54
工具和资源 ..... 45
多账户策略 ..... 128
为多租户集群规划多工作负载账户策略 ..... 128
集中式 EKS 集群 ..... 128

De-centralized EKS 集群 ..... 134
集中集群与 De-centralized EKS 集群 ..... 136
集群访问管理 ..... 137
EKS 访问管理选项 ..... 138
选项 1：带有集群访问管理 (CAM) API 的 AWS IAM 身份中心 ..... 138
选项 2：AWS IAM Users/Roles 映射到 Kubernetes 群组 ..... 140
选项 3：OIDC 提供商 ..... 140
适用于工作负载的 AWS EKS Pod 身份与 IRSA ..... 141
建议 ..... 141
集群自动扩展 ..... 143
EKS 自动模式 ..... 143
使用自动模式的理由 ..... 144
常见问题解答 ..... 144
Karpenter ..... 149
建议 ..... 54
Karpenter 最佳实践 ..... 150
正在创建 NodePools ..... 153
调度 Pod ..... 157
CoreDNS 推荐 ..... 159
Karpenter 蓝图 ..... 159
其他资源 ..... 79
Cluster Autoscaler ..... 160
概述 ..... 160
针对性能和可扩展性进行优化 ..... 164
针对成本和可用性进行优化 ..... 166
高级使用案例 ..... 169
其他参数 ..... 170
其他资源 ..... 79
参考 ..... 71
可靠性 ..... 174
如何使用本指南 ..... 3
简介 ..... 1
反馈 ..... 5
应用程序 ..... 176
建议 ..... 54
水平吊舱自动扩缩器 (HPA) ..... 180

垂直吊舱自动扩缩器 (VPA) ..... 181
更新应用程序 ..... 181
Health 检查和自我修复 ..... 183
建议 ..... 184
应对中断 ..... 185
建议 ..... 186
可观测性 ..... 187
建议 ..... 188
控制层面 ..... 190
EKS 架构 ..... 190
建议 ..... 191
监控控制平面指标 ..... 191
集群认证 ..... 194
入学网络挂钩 ..... 196
处理集群升级 ..... 197
集群终端节点连接 ..... 197
运行大型集群 ..... 198
其他资源： ..... 198
数据层面 ..... 198
建议 ..... 54
Networking ..... 204
Kubernetes 网络模型 ..... 204
容器网络接口 (CNI) ..... 205
亚马逊虚拟私有云 (VPC) Virtual Private Cloud CNI ..... 205
子网计算器 ..... 206
VPC 和子网注意事项 ..... 207
概述 ..... 160
建议 ..... 54
Amazon VPC CNI ..... 218
概述 ..... 160
建议 ..... 54
优化 IP 地址利用率 ..... 230
优化节点级 IP 消耗 ..... 230
减轻 IP 耗尽的风险 ..... 230
正在运行 IPv6 集群 ..... 238
概述 ..... 160

建议 ..... 54
自定义网络 ..... 254
示例配置 ..... 255
建议 ..... 54
适用于 Linux 的前缀模式 ..... 261
建议 ..... 54
适用于 Windows 的前缀模式 ..... 266
建议 ..... 54
每个 Pod 的安全组 ..... 271
建议 ..... 54
负载均衡 ..... 279
选择 Load Balancer 类型 ..... 279
配置负载均衡器 ..... 280
选择 Load Balancer Target-Type ..... 281
配置 Load Balancer 运行状况检查 ..... 283
可用性和 Pod 生命周期 ..... 284
参考 ..... 71
附录 ..... 288
监控网络性能问题 ..... 289
监控 CoreDNS 流量是否存在 DNS 限制问题 ..... 289
使用 Conntrack 指标监控 DNS 查询延迟 ..... 289
其他重要的网络性能指标 ..... 290
捕获指标以监控工作负载是否存在网络性能问题 ..... 290
在 IPVS 模式下运行 kube-proxy ..... 298
概述 ..... 160
可扩展性 ..... 302
如何使用本指南 ..... 302
了解缩放维度 ..... 302
超大缩放比例 ..... 303
控制层面 ..... 303
限制工作负载和节点爆发 ..... 304
安全地向下扩展节点和 pod ..... 304
运行 Kubectl 时使用 Client-Side 缓存 ..... 305
禁用 kubectl 压缩 ..... 305
分片集群自动扩缩器 ..... 306
API 优先级和公平性 ..... 307

在 API 服务器中检索资源 ..... 313
数据层面 ..... 319
自动节点自动缩放 ..... 319
使用许多不同的 EC2 实例类型 ..... 319
最好使用更大的节点来减少 API 服务器的负载 ..... 320
使用相似的节点大小以获得一致的工作负载性能 ..... 321
高效使用计算资源 ..... 322
自动更新亚马逊系统映像 (AMI) ..... 322
为容器使用多个 EBS 卷 ..... 322
如果工作负载使用 EBS 卷，请避开 EBS 连接限制较低的实例 ..... 324
禁用不必要的磁盘日志记录 ..... 324
当需要操作系统更新速度时，将实例修补到位 ..... 325
集群服务 ..... 325
扩展 CoreDNS ..... 325
垂直扩展 Kubernetes 指标服务器 ..... 327
CoreDNS lameduck 时长 ..... 327
CoreDNS 就绪情况调查 ..... 327
记录和监视代理 ..... 328
工作负载 ..... 329
使用 IPv6 进行容器联网 ..... 329
限制每个命名空间的服务数量 ..... 329
了解 Elastic 负载均衡器配额 ..... 329
使用 Route 53、Global Accelerator 或 CloudFront ..... 330
使用 EndpointSlices 代替终端节点 ..... 330
如果可能，请使用不可变和外部机密 ..... 331
限制部署历史记录 ..... 331
默认情况下禁用 “ServiceLinks 启用” ..... 332
限制每个资源的动态准入 webhook ..... 332
比较多个集群中的工作负载 ..... 332
缩放背后的理论 ..... 333
节点与流失率 ..... 333
每秒查询次数中的思考 ..... 333
扩展分布式组件 ..... 334
上游和下游瓶颈 ..... 335
按指标缩放 ..... 336
控制平面监控 ..... 338

API 服务器 ..... 338
问题出在哪里？ ..... 339
调度器 ..... 344
Kube 控制器管理器 ..... 345
ETCD ..... 347
节点效率和扩展 ..... 348
节点选择 ..... 348
节点 Bin-packing ..... 350
利用率与饱和度 ..... 357
设置 CPU 限制 ..... 365
Kubernetes SLO ..... 367
Kubernetes SLO ..... 368
Kubernetes SLI 指标 ..... 370
集群上的 SLO ..... 373
已知限制和 Service Quotas ..... 374
其他 AWS Service Quotas ..... 374
AWS 请求限制 ..... 379
其他已知限制 ..... 379
集群升级 ..... 381
概述 ..... 381
升级之前 ..... 381
让您的集群保持最新状态 ..... 382
查看 EKS 发布日历 ..... 382
了解分担责任模式如何适用于集群升级 ..... 383
就地升级集群 ..... 383
按顺序升级您的控制平面和数据平面 ..... 383
使用 EKS 文档创建升级清单 ..... 384
使用 Kubernetes API 升级插件和组件 ..... 384
升级前验证 EKS 的基本要求 ..... 385
验证可用的 IP 地址 ..... 386
验证 EKS 的 IAM 角色 ..... 386
迁移到 EKS Add-ons ..... 387
在升级控制平面之前，识别并修复已删除的 API 使用情况 ..... 388
集群见解 ..... 388
Kube-no-trouble ..... 390
冥王星 ..... 391

资源 ..... 391
更新 Kubernetes 工作负载。使用 kubectl-convert 更新清单 ..... 392
配置 PodDisruptionBudgets 和拓扑结构SpreadConstraints 以确保升级数据平面时工作负载的可用性 ..... 392
使用托管节点组或 Karpenter 来简化数据平面升级 ..... 394
确认版本与现有节点和控制平面的兼容性 ..... 394
为 Karpenter 托管的节点启用节点到期时间 ..... 395
对 Karpenter 托管的节点使用漂移功能 ..... 395
使用 eksctl 自动升级自管理节点组 ..... 395
升级前备份集群 ..... 396
升级控制平面后重启 Fargate 部署 ..... 396
评估 Blue/Green 集群作为就地集群升级的替代方案 ..... 396
跟踪 Kubernetes 项目中计划中的重大变更——提前思考 ..... 397
移除功能的具体指南 ..... 397
在 1.25 中移除 Dockershim ——在 Docker 插槽中使用探测器 (DDS) ..... 397
PodSecurityPolicy 在 1.25 中移除-迁移到 Pod 安全标准或策略即代码解决方案 ..... 397
1.23 中弃用 In-Tree 存储驱动程序-迁移到容器存储接口 (CSI) 驱动程序 ..... 398
其他资源 ..... 398
ClowdHaus EKS 升级指南 ..... 398
GoNoGo ..... 398
成本优化 ..... 399
一般指导原则 ..... 399
EKS 成本优化最佳实践 ..... 399
如何使用本指南 ..... 3
AWS 服务和 Kubernetes 的主要功能 ..... 400
反馈 ..... 5
框架 ..... 400
See 支柱：衡量和问责制 ..... 401
节省支柱：成本优化 ..... 401
计划支柱：规划和预测 ..... 402
跑步支柱 ..... 402
参考 ..... 71
意识 ..... 403
建议 ..... 54
其他工具 ..... 410
计算 ..... 411

正确调整工作负载规模 ..... 411
减少消费 ..... 412
减少未使用的容量 ..... 413
优化计算容量类型 ..... 417
优化计算使用率 ..... 420
Network ..... 421
Pod 到 Pod 通信 ..... 422
Load Balancer 到 Pod 通信 ..... 433
从容器注册表传输数据 ..... 435
将数据传输到互联网和 AWS 服务 ..... 436
之间的数据传输 VPCs ..... 438
使用服务网格 ..... 440
其他资源 ..... 79
存储 ..... 448
概览 ..... 160
短暂卷 ..... 448
永久卷 ..... 449
其他考虑因素 ..... 454
可观察性 ..... 454
简介 ..... 454
日志记录 ..... 455
EKS 控制飞机 ..... 455
EKS 数据平面 ..... 456
Metrics ..... 459
跟踪 ..... 462
其他资源： ..... 79
Windows ..... 465
AMI 管理 ..... 465
管理你自己的亚马逊 EKS 优化的 Windows AMI ..... 466
为自定义 EKS 优化的 AMI 配置更快的启动 ..... 467
在自定义 AMI 上缓存 Windows 基础图层 ..... 467
博客文章 ..... 469
适用于 Windows 容器的 GMSA ..... 469
什么是 GMSA 账户 ..... 469
Windows 容器和 GMSA 用例 ..... 469
Windows 服务器强化 ..... 471

使用 Windows 服务器核心减少攻击面 ..... 471
避开 RDP 连接 ..... 472
Amazon Inspector ..... 473
Amazon GuardDuty ..... 474
适用于 Windows 的 Amazon EC2 中的安全 ..... 474
正在扫描 Windows 镜像 ..... 474
Windows 版本和许可 ..... 475
Windows Server 版本 ..... 475
许可 ..... 475
日志记录 ..... 475
记录建议 ..... 476
监控 Windows 容器 ..... 477
Windows 联网 ..... 481
Windows 容器网络概述 ..... 481
IP 地址管理 ..... 482
容器网络接口 (CNI) 选项 ..... 484
网络策略 ..... 485
内存和系统管理 ..... 485
预留系统和 kubelet 内存 ..... 485
Windows 容器内存要求 ..... 486
结论 ..... 486
基础设施管理 ..... 486
推送和拉取 Windows 镜像 ..... 487
参考 ..... 489
计划 ..... 489
将 POD 分配给节点最佳实践 ..... 489
确保 OS-specific 工作负载落在相应的容器主机上 ..... 489
在同一个集群中处理多个 Windows 构建 ..... 490
Pod 清单中的简 NodeSelector 化和容忍度使用 RuntimeClass ..... 492
托管节点组 Support ..... 493
其他文档 ..... 493
Windows 容器的 Pod 安全 ..... 493
存储选项 ..... 496
什么是树内和树外音量插件？ ..... 496
In-tree 适用于 Windows 的音量插件 ..... 496
Out-of-tree 适用于 Windows ..... 500

适用于 Windows File Server 的 Amazon FSx ..... 500
强化 Windows 容器镜像 ..... 501
1. 使用本地安全策略和注册表配置账户策略（密码或锁定） ..... 502
2. 审计策略 ..... 504
3. Windows 容器的 IIS 安全最佳实践 ..... 506
4. 最低特权原则 ..... 510
最后的想法：为什么保护你的 Windows 容器是当今威胁格局 Must-Have 中的一个问题 ..... 512
Hybrid ..... 514
网络断开连接 ..... 514
最佳实践 ..... 514
Kubernetes 容器故障转移 ..... 519
应用程序网络流量 ..... 524
主机凭证 ..... 528
AI/ML ..... 530
反馈 ..... 530
计算 ..... 530
GPU 资源优化和成本管理 ..... 530
节点弹性和训练 Job 管理 ..... 541
应用程序扩展和性能 ..... 543
用于高级 GPU 管理的动态资源分配 ..... 545
CPU 推理 ..... 589
概述 ..... 160
为什么 CPU 用于 AI 工作负载？ ..... 591
CPU 与 GPU /Trainium：何时选择每个 ..... 591
工作负载决策框架 ..... 592
生产模式 ..... 594
优化最佳实践 ..... 595
评估 CPU-First 工作负载的模型质量 ..... 600
参考 ..... 71
Networking ..... 602
对于 Inter-Node 通信量大的应用程序，可以考虑使用更高的网络带宽或弹性结构适配器 ..... 602
规划大型 GPU 实例的 IP 地址消耗 ..... 602
安全性 ..... 603
安全性与合规性 ..... 603
仓储服务 ..... 604
数据管理和存储 ..... 604

可观测性 ..... 614
监控和可观测性 ..... 615
可观察性和指标 ..... 616
性能 ..... 628
管理 ML 构件、提供框架和启动优化 ..... 628
优化容器镜像提取时间 ..... 630
考虑将 NVMe 用于 kubelet 和 containerd 存储 ..... 633
改进 ..... 634
现有贡献者摘要 ..... 634
设置本地编辑环境 ..... 634
分叉并克隆存储库 ..... 634
打开 VS 代码工作区 ..... 634
编辑文件 ..... 635
提交拉取请求 ..... 635
使用 github.dev 基于网络的编辑器 ..... 635
编辑单个页面 ..... 636
查看和设置页面的 ID ..... 636
设置页面 ID ..... 636
创建新页面 ..... 637
创建页面元数据 ..... 637
添加到目录 ..... 637
插入图片 ..... 638
向淡水河谷查询风格 ..... 638
生成本地预览版 ..... 639
AsciiDoc 备忘单 ..... 639
基本格式化 ..... 639
标头 ..... 639
Lists ..... 639
链接 ..... 640
图片 ..... 640
代码块 ..... 640
表 ..... 640
警戒插件 ..... 641
包含 ..... 641
..... dcxlii

# 亚马逊 EKS 最佳实践指南

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

欢迎来到 EKS 最佳实践指南。该项目的主要目标是为 Amazon EKS 的第二天运营提供一套最佳实践。我们选择将本指南发布到， GitHub 这样我们就可以快速进行迭代，针对各种问题提供及时有效的建议，并轻松采纳来自更广泛社区的建议。

我们目前已经发布了以下主题的指南：

- 安全最佳实践
- 可靠性最佳实践
- 集群自动扩缩的最佳实践：Karpenter
- 集群自动扩缩的最佳实践：集群自动扩缩器
- 集群自动扩缩的最佳实践：EKS 自动模式
- 联网最佳实践
- 可扩展性最佳实践
- 群集升级的最佳实践
- 成本优化的最佳实践
- 运行 Windows 容器的最佳实践
- 混合部署的最佳实践
- 运行 AI/ML 工作负载的最佳实践

我们还开源了一个名为h ardenek s的基于Python的CLI（命令行界面），以查看本指南中的一些建议。

未来，我们将发布有关绩效、成本优化和卓越运营的最佳实践指南。

## 相关指南

除了 EKS 用户指南外，AWS 还发布了其他几份指南，可以帮助您实施 EKS。

- EMR 容器最佳实践指南

- EKS 上的数据
- AWS 可观察性最佳实践
- 适用于 Terraform 的 Amazon EKS 蓝图
- 亚马逊 EKS 蓝图快速入门

# 贡献

我们鼓励您为这些指南做出贡献。如果您实施的做法已被证明行之有效，请通过提出问题或拉取请求与我们分享。同样，如果您在我们已经发布的指南中发现了错误或缺陷，请提交 PR 进行更正。提交 PR 的指南可以在我们的贡献指南中找到。

# 安全最佳实践

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

本指南提供有关保护依赖于 EKS 的信息、系统和资产的建议，同时通过风险评估和缓解策略实现业务价值。此处的指南是 AWS 发布的一系列最佳实践指南的一部分，旨在帮助客户根据最佳实践实施 EKS。性能、卓越运营、成本优化和可靠性指南将在未来几个月内发布。

## 如何使用本指南

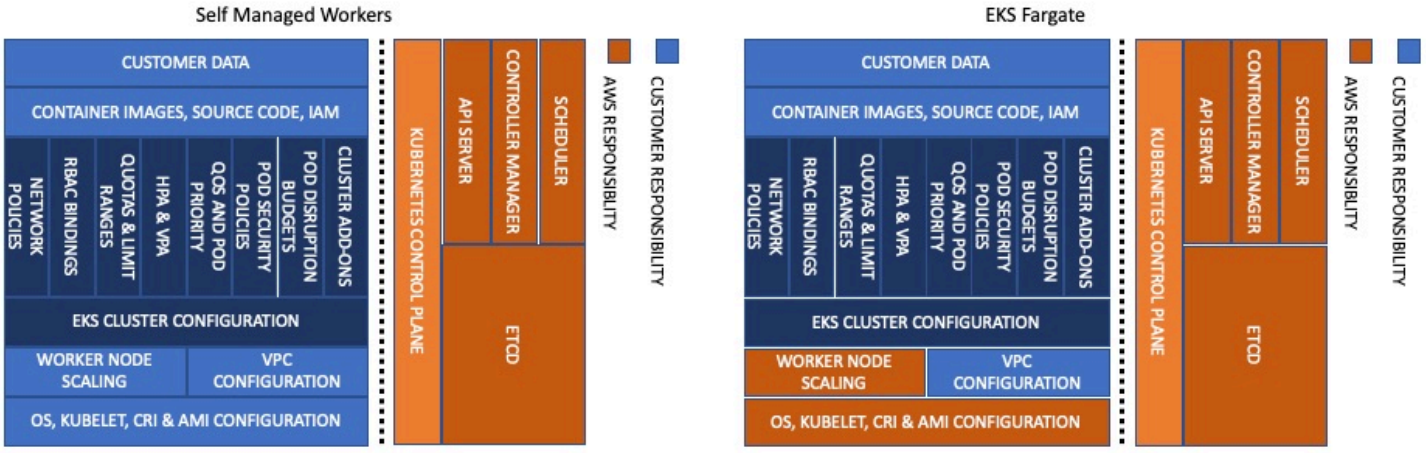
本指南适用于负责实施和监控 EKS 集群及其支持的工作负载安全控制的有效性的安全从业人员。为了便于阅读，该指南分为不同的主题领域。每个主题都以简要概述开头，然后列出保护 EKS 集群的建议和最佳实践。无需按特定顺序阅读主题。

## 了解分担责任模型

使用像 EKS 这样的托管服务时，安全性和合规性被视为共同责任。一般而言，AWS 负责云的 “安全”，而您（客户）则负责云 “中” 的安全。在 EKS 中，AWS 负责管理 EKS 托管的 Kubernetes 控制平面。这包括 Kubernetes 控制平面节点、ETCD 数据库以及 AWS 提供安全可靠服务所需的其他基础设施。作为 EKS 的消费者，您对本指南中的主题负有主要责任，例如 IAM、pod 安全、运行时安全、网络安全等。

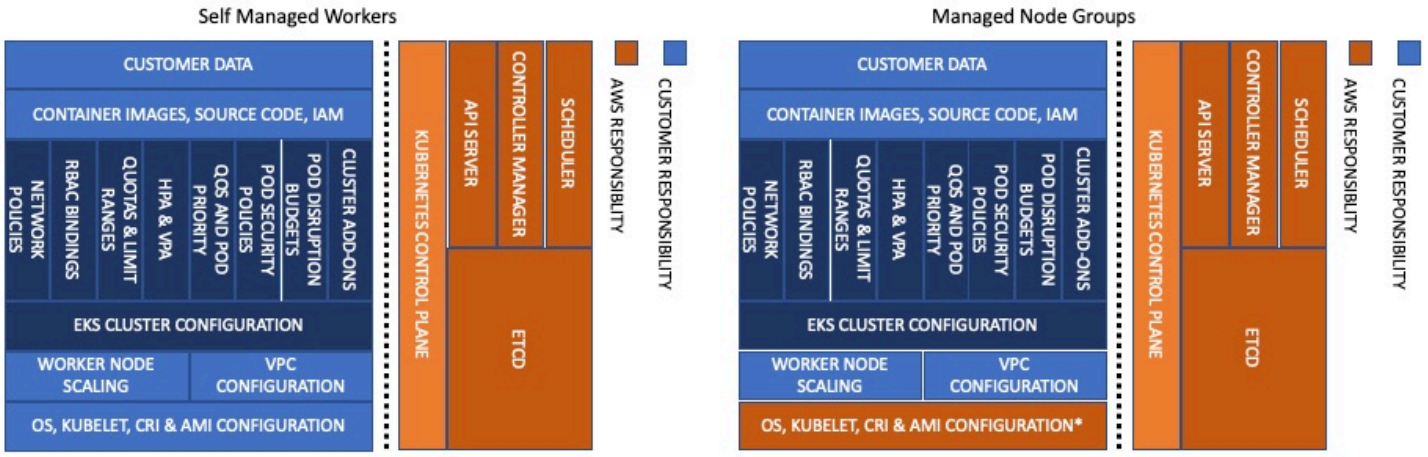
在基础设施安全方面，当您从自我管理的工作人员转移到托管节点组再到Fargate时，AWS将承担额外的责任。例如，在 Fargate 中，AWS 负责保护 instance/runtime 用于运行您的 Pod 的底层。

责任共担模型——Fargate




AWS还将承担责任，通过Kubernetes补丁版本和安全补丁使经过EKS优化的AMI保持最新状态。使用托管节点组 (MNG) 的客户有责任通过 EKS API、CLI、Cloudformation 或 AWS 控制台将其节点组升级到最新 AMI。同样与 Fargate 不同，MNG 不会自动缩放你的规模。infrastructure/cluster这可以通过集群自动扩缩器或其他技术来处理，例如Kar pent er、AWS原生自动缩放 SpotInst、Ocean或Atlas sian的自动扶梯。

责任共担模型-MNG




在设计系统之前，重要的是要知道您的责任与服务提供商 (AWS) 之间的分界线。

有关责任共担模式的更多信息，请参阅 https://aws.amazon.com/compliance/shared-responsibility-model/

# 简介

在使用像 EKS 这样的托管 Kubernetes 服务时，有几个安全最佳实践领域是相关的：

- 身份和访问管理
- Pod 安全
- 运行时安全
- 网络安全性
- Multi-tenancy
- 多账户适用于 Multi-tenancy
- 侦探控制
- 基础设施安全性
- 数据加密和机密管理
- 监管合规
- 事件响应和取证
- 图像安全

在设计任何系统时，您都需要考虑其安全影响以及可能影响您的安全状况的做法。例如，您需要控制谁可以对一组资源执行操作。您还需要能够快速识别安全事件，保护您的系统和服务免受未经授权的访问，并通过数据保护来维护数据的机密性和完整性。拥有一套定义明确且经过演练的流程来应对安全事件，也将改善您的安全状况。这些工具和技术非常重要，因为它们有助于实现诸如避免财务损失或履行监管义务等目标。

AWS 通过提供丰富的安全服务来帮助组织实现其安全和合规目标，这些服务是根据广大注重安全的客户的反馈而演变而来的。通过提供高度安全的基础，客户可以将更少的时间花在 “无差别的繁重工作” 上，而将更多的时间花在实现其业务目标上。

# 反馈

本指南发布的目的是 GitHub 收集来自更广泛 EKS/Kubernetes 社区的直接反馈和建议。如果您认为我们应该在指南中包含最佳实践，请在 GitHub 存储库中提交问题或提交 PR。我们的意图是在服务中添加新功能或出现新的最佳实践时，定期更新指南。

# 阅读更多内容

Kubernetes 安全白皮书由安全审计工作组赞助，描述了 Kubernetes 攻击面和安全架构的关键方面，旨在帮助安全从业人员做出合理的设计和实施决策。

CNCF 还发布了一份关于云原生安全的白皮书。该论文探讨了技术格局是如何演变的，并主张采用与 DevOps 流程和敏捷方法相一致的安全实践。

## 工具和资源

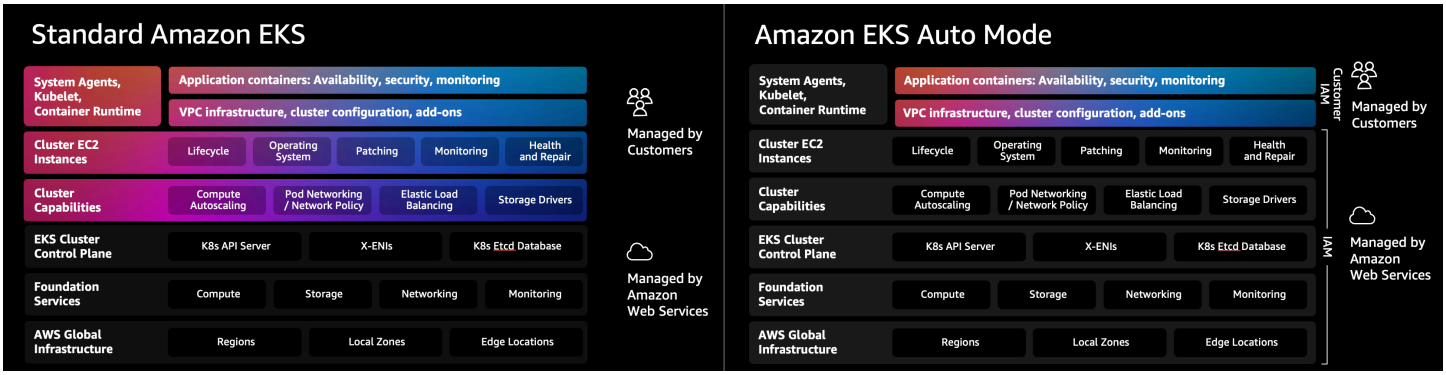
Amazon EKS 安全沉浸式研讨会

## EKS 自动模式-安全

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Amazon EKS 自动模式通过将 AWS 的安全管理扩展到控制平面之外，将工作节点和核心集群组件包括在内，从而引入了增强的安全功能。这种全面的安全模型可帮助组织保持强大的安全态势，同时减少运营开销。

责任共担模型-EKS 自动模式




EKS 自动模式中的主要安全增强功能包括：

- 最小的容器优化操作系统，减少了攻击面
- 通过 EC2 托管实例强制实施安全最佳实践
- 通过强制节点轮换实现自动安全补丁管理
- Built-in 节点隔离和安全边界
- 简化了 IAM 集成，所需权限最少

默认情况下，EKS Auto Mode 会强制执行这些安全控制，帮助组织满足其安全和合规性要求，同时简化集群操作。这种方法符合深度防御原则，在基础架构、节点和工作负载级别提供多层安全控制。

> **Important**
>
> 虽然 EKS Auto Mode 提供了增强的安全功能，但组织仍应在应用程序层实施适当的安全控制，并遵循集群上运行的工作负载的安全最佳实践。

## 安全架构

EKS Auto Mode 在 EKS 基础设施的多个层面上实施安全控制，从控制平面到单个节点。了解这种架构对于有效管理和保护 EKS 集群至关重要。

### 控制平面安全

EKS 自动模式下的 EKS 控制平面保持与传统 EKS 集群相同的高安全标准，同时增加了新的安全功能：

- 信封加密：所有 Kubernetes API 数据都使用信封加密自动加密。
- KMS 集成：将 AWS KMS 与 Kubernetes KMS 提供商 v2 一起使用，并提供 AWS-owned 密钥或客户托管密钥 (CMK) 选项。
- 增强的组件管理：auto-scaling、ENI 管理和 EBS 控制器等关键组件被移出集群并由 AWS 管理。
- 改进的安全控制和审计功能：除了标准 EKS 集群之外，EKS Auto Mode 所需的权限完全通过集群 IAM 角色而不是单个节点角色进行管理。

### IAM 集成和访问管理

EKS 自动模式通过 EKS 访问条目和 EKS Pod 身份增强了与 AWS 身份和访问管理 (IAM) 的集成。

#### 集群访问管理

EKS 自动模式通过集群访问管理 (CAM) API 引入了对集群访问管理的改进：

- 通过以下方式实现标准化身份验证模式 `EKS_API`
- 通过 API-based 访问管理增强安全性
- 使用访问条目和访问策略简化访问控制

可以创建访问条目来管理集群访问权限：

```
aws eks create-access-entry \
    --cluster-name ${EKS_CLUSTER_NAME} \
    --principal-arn arn:aws:iam::${ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
    --type STANDARD
```

> **Important**
>
> 虽然仍然可以创建具有CONFIG_MAP_AND_API身份验证模式的 EKS Auto Mode 集群，但这不是标准方法，强烈建议对新集群使用默认API身份验证模式。 API与传统 ConfigMap-based 方法相比，基于身份验证提供了增强的安全性并简化了访问管理。

### EKS 容器组身份

EKS 自动模式附带已部署的 Pod 身份代理，从而可以简化向 Pod 授予 AWS IAM 权限的方法：

- 无需配置 OIDC 提供商即可简化 IAM 权限管理
- 与 IRSA 相比，减少了运营开销
- 通过会话标记和 ABAC 支持增强安全性

```
aws eks create-pod-identity-association \
  --cluster-name ${EKS_CLUSTER_NAME} \
  --role-arn arn:aws:iam::${AWS_ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
  --namespace ${NAMESPACE} \
  --service-account ${SERVICE_ACCOUNT_NAME}
```

> **Important**
>
> Pod Identity 是在 EKS 自动模式下获取 IAM 权限的推荐方法，因为与 IRSA 相比，它提供了增强的安全功能和简化的管理。

### 节点 IAM 角色

EKS 自动模式使用新的模式AmazonEKSWorkerNodeMinimalPolicy，该模式仅提供 EKS 自动模式节点运行所需的权限。这些权限：

- 与传统节点策略相比，提供的权限集更少
- 坚持最低权限原则
- 由 AWS 自动管理和更新

这种最低限度的策略方法通过限制节点及其工作负载的可用权限来帮助改善安全状况。

## 节点安全

EKS 自动模式在节点级别引入了几项重要的安全改进：

### EC2 托管实例安全

EKS 自动模式节点使用具有增强安全属性的 Amazon EC2 托管实例：

- IAM-enforced 限制措施可防止可能损害 AWS 操作节点能力的操作
- 不可变的基础架构模式，其中配置更改需要更换节点
- 必须在 21 天内更换节点，以确保定期进行安全更新
- 使用 imdsv2 限制访问实例元数据，并限制跳跃限制

### 操作系统安全性

该操作系统是 Bottlerock et 的自定义变体，针对 EKS 自动模式进行了优化，其中包括：

- Read-only 根文件系统
- SELinux 默认启用，并具有强制访问控制
- 使用独特的 SELinux MCS 标签自动隔离 Pod
- 已禁用 SSH 访问并移除不必要的服务
- 通过节点轮换自动修补安全补丁

### 节点组件安全

节点组件配置了安全最佳实践：

- Kubelet 配置了安全的默认值
- 容器运行时强化配置
- 自动证书管理和轮换

- 受限的节点到控制平面的通信

## 网络安全性

EKS Auto Mode 实现了多项网络安全功能，以确保集群内以及与外部资源的安全通信：

VPC CNI 网络策略

EKS 自动模式利用了 Amazon VPC CNI 插件的原生 Kubernetes 网络策略支持：

- 与上游 Kubernetes 网络策略 API 集成
- 允许对 pod 到 Pod 的通信进行精细控制
- 同时支持入口和出口规则

要在 EKS 自动模式下启用网络策略支持，您需要使用`configMap`清单配置 VPC CNI 插件。示例如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: amazon-vpc-cni
  namespace: kube-system
data:
  enable-network-policy: "true"
```

还需要定义在 Node Class 中配置的网络策略支持，如下所示：

```
apiVersion: eks.amazonaws.com/v1
kind: NodeClass
metadata:
  name: example-node-class
spec:
  networkPolicy: DefaultAllow
  networkPolicyEventLogs: Enabled
```

启用后，您可以创建网络策略来控制流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

#### 增强的 ENI 管理

EKS 自动模式为弹性网络接口 (ENI) 管理提供了更高的安全性：

- AWS-managed ENI 连接和配置
- 将控制流量与数据流量分开
- 自动管理 IP 地址，降低了节点所需的权限

## 存储安全

EKS 自动模式为临时存储和永久存储提供了增强的安全功能：

#### 临时存储

- 写入临时卷的所有数据都会自动加密
- 使用行业标准的 AES-256 加密算法
- 该服务可无缝处理加密和解密

#### EBS 卷

- 根卷和数据 EBS 卷始终处于加密状态
- 卷配置为在实例终止时删除
- 可以选择指定用于加密的自定义 KMS 密钥

#### EFS 集成

- 支持通过 EFS 传输中的加密
- EFS 文件系统的静态自动加密
- 与 EFS 接入点集成以增强访问控制

> **Important**
>
> 在 EKS 自动模式下使用 EFS 时，请确保在 EFS 文件系统级别配置相应的加密设置，因为 EKS 自动模式不会直接管理 EFS 加密。

## 监控和日志记录

EKS Auto Mode 提供了增强的监控和日志记录功能，可帮助您保持对集群安全状况和运行状况的可见性。

### 控制层面日志记录

EKS Auto Mode 保持与标准 EKS 相同的控制平面日志功能，但是它默认启用所有日志以增强监控。

- 日志将发送到 Amazon CloudWatch 日志
- 默认情况下，EKS 自动模式启用所有控制平面日志：API 服务器、审计、身份验证器、控制器管理器和调度器
- EKS 自动模式可实现对集群操作和安全事件的详细可见性

> **Important**
>
> 控制平面日志记录会产生额外的日志存储成本。 CloudWatch请仔细考虑您的日志策略，以便在安全需求和成本管理之间取得平衡。

### Node-level 正在记录

EKS 自动模式增强了节点级日志记录：

- 系统日志是自动收集的，可以通过 CloudWatch 日志进行访问
- 即使在节点终止后仍会保留节点日志，这有助于事后分析
- 增强了对节点级安全事件和操作问题的可见性

## 亚马逊 GuardDuty 集成

EKS 自动模式集群与 Amazon 无缝集成 GuardDuty ，以增强威胁检测。功能包括：

- 自动扫描控制平面审计日志
- 可以为工作负载监控启用的运行时监控
- 与现有 GuardDuty 发现和警报机制集成

要在 Amazon GuardDuty 上为 Kubernetes 审计日志启用 EKS 自动模式保护，你可以运行以下命令：

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{"Kubernetes":{"AuditLogs":{"Enable":true}}}'
```

运行时安全的 Amazon GuardDuty 集成

Amazon 为 EKS 自动模式集群 GuardDuty 提供基本的运行时安全监控，提供全面的威胁检测和安全监控功能。这种集成尤其重要，因为它有助于实时识别潜在的安全威胁和恶意活动。

EKS 自动模式的主要 GuardDuty 功能

- 运行时监控：
  - 持续监控运行时行为
  - 检测恶意或可疑活动
  - 识别潜在的集装箱逃生企图
  - 监控异常流程执行或网络连接
- Kubernetes-Specific 威胁检测：
  - 识别可疑的 pod 部署尝试
  - 检测受损的容器
  - 监控特权容器启动
  - 识别可疑的服务账户使用情况
- 综合发现类型：
  - Policy:Kubernetes/*-检测违反安全最佳实践的行为
  - Impact:Kubernetes/*-识别可能受影响的资源
  - Discovery:Kubernetes/*-检测侦察活动
  - Execution:Kubernetes/*-识别可疑的执行模式
  - Persistence:Kubernetes/*-检测潜在的持续威胁

要在 Amazon GuardDuty 上为 Kubernetes 审计日志和运行时监控启用 EKS 自动模式保护，您可以运行以下命令：

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{
        "Kubernetes": {
            "AuditLogs": {"Enable": true},
            "RuntimeMonitoring": {"Enable": true}
        }
    }'
```

**Important**

GuardDuty EKS Auto Mode 集群自动支持运行时监控，无需在节点级别进行额外配置即可提供增强的安全可见性。

GuardDuty 调查结果整合

GuardDuty 调查结果可以与其他 AWS 服务集成以实现自动响应：

- EventBridge 规则：

```
{
  "source": ["aws.guardduty"],
  "detail-type": ["GuardDuty Finding"],
  "detail": {
    "type": ["Runtime:Container/*", "Runtime:Kubernetes/*"],
    "severity": [4, 5, 6, 7, 8]
  }
}
```

- S@@ ecurity Hub 集成：

```
# Enable Security Hub integration
aws securityhub enable-security-hub \
    --enable-default-standards \
    --tags '{"Environment":"Production"}' \
```

```
    --region us-west-2
```

GuardDuty 使用 EKS 自动模式的最佳实践

1. 启用所有查找类型：
   - 同时启用 Kubernetes 审计日志监控和运行时监控
   - 为所有严重性级别配置调查结果
2. 实现自动响应：
   - 为高严重性发现创建 EventBridge 规则
   - 与 AWS Security Hub 集成，实现集中式安全管理
   - 在适当的情况下设置自动补救措施
3. 定期回顾和调整：
   - 定期审查 GuardDuty 调查结果
   - 根据您的环境调整检测阈值
   - 根据新的查找结果类型更新响应程序
4. Cross-Account 管理：
   - 考虑使用 GuardDuty 管理员帐户进行集中管理
   - 启用跨多个账户的调查结果汇总

> **Warning**
>
> 虽然 GuardDuty 提供全面的安全监控，但它应该是深度防御策略的一部分，该策略包括其他安全控制措施，例如网络策略、Pod 安全标准和适当的 RBAC 配置。

## 常见问题 (FAQ)

问：EKS 自动模式在安全性方面与标准 EKS 有何不同？ 答：EKS 自动模式通过 EC2 托管实例、自动修补、强制节点轮换和内置安全控制来增强安全性。通过让 AWS 管理更多安全方面，它减少了运营开销，同时保持了强大的安全态势。

问：我还能在 EKS 自动模式下使用现有的安全工具和策略吗？ 答：是的，EKS 自动模式与大多数现有的安全工具和策略兼容。但是，由于 EKS Auto Mode 节点的托管性质，某些节点级安全工具可能需要调整。

问：如何在 EKS 自动模式下部署安全代理和监控工具？ 答：在 EKS 自动模式下，安全代理和监控工具应作为 Kubernetes 工作负载部署（通常 DaemonSets默认在每个节点上部署一个 Pod 实例），而不是直接安装在节点操作系统上。这种方法与 EKS 自动模式的不可变基础架构模型一致。示例：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: security-agent
  namespace: security
spec:
  selector:
    matchLabels:
      app: security-agent
  template:
    metadata:
      labels:
        app: security-agent
    spec:
      containers:
      - name: security-agent
        image: security-vendor/agent:latest
        securityContext:
          privileged: false
          # Use specific capabilities instead of privileged mode
          capabilities:
            add: ["NET_ADMIN", "SYS_ADMIN"]
```

问：第三方安全解决方案是否与 EKS 自动模式兼容？ 答：许多流行的第三方安全解决方案已更新为支持 EKS Auto Mode，但始终建议您向您的安全供应商验证具体版本和部署要求，因为支持 EKS Auto Mode 可能需要更新版本或特定的部署配置。

问：在 EKS 自动模式下，安全代理有哪些限制？ 答：主要限制包括：

- 无法直接访问修改节点的操作系统
- 节点轮换之间没有持久性
- 必须与基于容器的部署兼容
- 需要尊重节点的不变性
- 可能需要不同的权限配置
- 对节点的任何持续更改都应通过NodePools和NodeClasses资源完成。

> **Note**
>
> 虽然 EKS Auto Mode 可能需要调整您的安全工具部署策略，但这些更改通常会使配置更易于维护和安全，符合云原生最佳实践。EKS 自动模式预计将完全接管其管理的大部分功能。因此，如果可以的话，您对这些功能所做的任何手动更改都可能被 EKS Auto Mode 覆盖或丢弃。

问：我能否在 EKS 自动模式下使用自定义 AMI？ 答：目前，EKS 自动模式不支持自定义 AMI。这是设计使然，因为 AWS 将节点的安全、修补和维护作为责任共担模式的一部分进行管理。EKS 自动模式节点使用 Bottlerocket 的特殊变体，该变体由 AWS 优化和维护。

问：在 EKS 自动模式下，节点多久自动旋转一次？ 答：EKS 自动模式下的节点最长生命周期为 21 天。在此限制之前，它们将被自动更换，从而确保定期进行安全更新和补丁应用程序。

问：我能否通过 SSH 连接到 EKS 自动模式节点进行故障排除？ 答：不可以，在 EKS 自动模式下不提供直接 SSH 访问。相反，您可以使用 NodeDiagnostic 自定义资源定义 (CRD) 来收集系统日志和调试信息。

问：在 EKS 自动模式下，是否默认启用网络策略支持？ 答：目前，需要通过 VPC CNI 插件配置明确启用网络策略支持。启用后，您可以使用标准的 Kubernetes 网络策略。

问：我应该在 EKS 自动模式下使用 IRSA 还是 Pod Identity？ 答：虽然两者都支持，但在 EKS 自动模式中推荐使用 Pod Identity 方法，因为它已经包含 Pod Identity Security 代理插件，并提供了增强的安全功能和简化的管理。

问：我还能在 EKS 自动模式下使用 aws-auth ConfigMap 吗？ 答：`aws-auth` ConfigMap 这是一项已弃用的功能。建议使用默认的 API-based 身份验证方法，以增强安全性并简化访问管理。

问：如何在 EKS 自动模式下监控安全事件？ 答：EKS 自动模式与多种监控解决方案集成 GuardDuty，包括 CloudWatch、和 CloudTrail。 GuardDuty 专门针对 EKS 工作负载提供增强的运行时安全监控。

问：如何从 EKS 自动模式节点收集日志？ 答：使用 NodeDiagnostic CRD，它会自动将日志上传到 S3 存储桶。您也可以将 Contain CloudWatch er Insights 和 AWS Distro 用于。 OpenTelemetry

> **Note**
>
> 随着新功能添加到 EKS Auto Mode 以及我们收到客户的常见问题，本常见问题解答部分会定期更新。

# 身份和访问管理

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Id@@ entity and Access Managem ent (IAM) 是一项 AWS 服务，它执行两个基本功能：身份验证和授权。身份验证涉及身份验证，而授权则控制 AWS 资源可以执行的操作。在 AWS 中，资源可以是其他 AWS 服务，例如 EC2，也可以是 AWS 委托人，例如 I AM 用户或角色。管理允许资源执行的操作的规则以 IAM 策略的形式表达。

## 控制 EKS 集群的访问权限

Kubernetes 项目支持各种不同的策略来验证对 kube-apiserver 服务的请求，例如不记名令牌、证书、OIDC 等。X.509 EKS 目前原生支持 webhook 令牌身份验证、服务账户令牌以及截至 2021 年 2 月 21 日的 OIDC 身份验证。

Webhook 身份验证策略调用一个 webhook 来验证不记名令牌。在 EKS 上，这些不记名令牌是在您运行命令时由 AWS CLI 或 aws-iam-authenticator 客户端生成的。`kubectl`当你执行命令时，令牌会被传递给 kube-apiserver，kube-apiserver 会将其转发到身份验证 webhook。如果请求格式正确，则 webhook 会调用嵌入在令牌正文中的预签名 URL。此 URL 验证请求的签名，并返回有关用户的信息，例如用户的账户、Arn 和 UserId kube-apiserver。

要手动生成身份验证令牌，请在终端窗口中键入以下命令：

```
aws eks get-token --cluster-name <cluster_name> --region <region>
```

输出应如下所示：

```
{
    "kind": "ExecCredential",
    "apiVersion": "client.authentication.k8s.io/v1alpha1",
    "spec": {},
    "status": {
        "expirationTimestamp": "2024-12-20T17:38:48Z",
        "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMudXMtd2VzdC0yLmFtYXpvbmF3cy5jb20vP0FjdGlvbj1HZ...."
```

```
    }
}
```

您也可以通过编程方式获取代币。以下是用 Go 编写的示例：

```
package main

import (
  "fmt"
  "log"
  "sigs.k8s.io/aws-iam-authenticator/pkg/token"
)

func main()  {
  g, _ := token.NewGenerator(false, false)
  tk, err := g.Get("<cluster_name>")
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(tk)
}
```

输出应如下所示：

```
{
  "kind": "ExecCredential",
  "apiVersion": "client.authentication.k8s.io/v1alpha1",
  "spec": {},
  "status": {
    "expirationTimestamp": "2020-02-19T16:08:27Z",
    "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMuYW1hem9uYXdzLmNvbS8_QWN0aW9uPUdldENhbGxlcklkZW50aXR5JlZlcnNpb249MjAxMS0wNi0x
  }
}
```

每个标记以开头，k8s-aws-v1.后跟一个 base64 编码的字符串。解码后的字符串应类似于以下内容：

```
https://sts.amazonaws.com/?Action=GetCallerIdentity&Version=2011-06-15&X-
Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=XXXXJPFRILKNSRC2W5QA
%2F20200219%2Fus-xxxx-1%2Fsts%2Faws4_request&X-Amz-Date=20200219T155427Z&X-
```

```
Amz-Expires=60&X-Amz-SignedHeaders=host%3Bx-k8s-aws-id&X-Amz-
Signature=XXXf8f3285e320ddb5e683a5c9a405301ad76546f24f28111fdad09cf648a393
```

该令牌由包含亚马逊凭证和签名的预签名 URL 组成。欲了解更多详情，请参阅 https://docs.aws.amazon.com/STS/latest/APIReference/API_GetCallerIdentity.html.

代币的存活时间 (TTL) 为 15 分钟，之后需要生成一个新代币。当您使用客户端时，会自动处理此问题`kubectl`，但是，如果您使用的是 Kubernetes 控制面板，则每次令牌到期时都需要生成新令牌并重新进行身份验证。

AWS IAM 服务对用户的身份进行身份验证后，kube-apiserver 就会读取`kube-system`命名空间`aws-auth` ConfigMap 中的内容，以确定要与该用户关联的 RBAC 组。用于在 `aws-auth` ConfigMap IAM 委托人（即 IAM 用户和角色）与 Kubernetes RBAC 组之间创建静态映射。可以在 Kub RoleBindings ernetes 中引用 RBAC 群组，或者。ClusterRoleBindings它们与 IAM 角色的相似之处在于，它们定义了一组可以对 Kubernetes 资源（对象）集合执行的操作（动词）。

## CloudWatch 查询可帮助用户识别向全局 STS 端点发送请求的客户端

运行下面的 CloudWatch 查询以获取 sts 端点。如果 stsendpoint 等于 “sts.amazonaws.com”，则它是全局 STS 端点。如果 stsendpoint 等于 “sts. <region>.amazonaws.com”，那么它就是一个区域 STS 终端节点。

```
fields @timestamp, @message, @logStream, @log,stsendpoint
| filter @logStream like /authenticator/
| filter @message like /stsendpoint/
| sort @timestamp desc
| limit 10000
```

## 集群访问管理器

集群访问管理器现在是管理 AWS IAM 委托人对 Amazon EKS 集群的访问的首选方式，它是 AWS API 的一项功能，也是 EKS v1.23 及更高版本集群（新集群或现有集群）的可选功能。它简化了 AWS IAM 和 Kubernetes RBAC 之间的身份映射，无需在 AWS 和 Kubernetes API 之间切换或编辑访问管理，从而减少了`aws-auth` ConfigMap 运营开销，并有助于解决配置错误。该工具还允许集群管理员撤销或完善自动授予用于创建集群的 AWS IAM 委托人的`cluster-admin`权限。

此 API 依赖于两个概念：

- 访问条目：直接关联到允许向 Amazon EKS 集群进行身份验证的 AWS IAM 委托人（用户或角色）的集群身份。

- 访问策略：是 Amazon EKS 的特定策略，它为访问条目提供在 Amazon EKS 集群中执行操作的授权。

在启动时，Amazon EKS 仅支持预定义策略和 AWS 托管策略。访问策略不是 IAM 实体，由 Amazon EKS 定义和管理。

集群访问管理器允许将上游 RBAC 与支持允许和通过（但不拒绝）有关 API 服务器请求的 Kubernetes AuthZ 决策的访问策略相结合。当上游 RBAC 和 Amazon EKS 授权机构都无法确定请求评估的结果时，就会做出拒绝决定。

借助此功能，Amazon EKS 支持三种身份验证模式：

1. CONFIG_MAP继续独占使用aws-auth配置地图。
2. API_AND_CONFIG_MAP从 EKS Access Entry API 和 ConfigMa aws-auth p 中获取经过身份验证的 IAM 委托人，优先考虑访问条目。非常适合将现有aws-auth权限迁移到访问条目。
3. API完全依赖 EKS Access Entry API。这是新推荐的方法。

首先，集群管理员可以创建或更新 Amazon EKS 集群，将首选身份验证设置为API_AND_CONFIG_MAP或API方法，并定义访问条目以授予所需的 AWS IAM 委托人访问权限。

```
$ aws eks create-cluster \
    --name <CLUSTER_NAME> \
    --role-arn <CLUSTER_ROLE_ARN> \
    --resources-vpc-config
 subnetIds=<value>,endpointPublicAccess=true,endpointPrivateAccess=true \
    --logging '{"clusterLogging":[{"types":
["api","audit","authenticator","controllerManager","scheduler"],"enabled":true}]}' \
    --access-config
 authenticationMode=API_AND_CONFIG_MAP,bootstrapClusterCreatorAdminPermissions=false
```

上面的命令是在没有集群创建者管理员权限的情况下创建 Amazon EKS 集群的示例。

可以使用命令更新 Amazon EKS 集群配置以启用 A API uthenticationMod update-cluster-config e，要使用该命令在现有集群上执行此操作，CONFIG_MAP则必须先更新到，API_AND_CONFIG_MAP然后再更新到。API这些操作无法恢复，这意味着无法从或切换API到API_AND_CONFIG_MAP或CONFIG_MAP，也无法从切换API_AND_CONFIG_MAP到CONFIG_MAP。

```
$ aws eks update-cluster-config \
```

```
    --name <CLUSTER_NAME> \
    --access-config authenticationMode=API
```

API 支持添加和撤消对集群的访问权限以及验证指定集群的现有访问策略和访问条目的命令。创建默认策略是为了匹配 Kubernetes RBAC，如下所示。

| EKS 访问政策 | Kubernetes RBAC |
| --- | --- |
| AmazonEKSClusterAdminPolicy | 集群管理员 |
| AmazonEKSAdminPolicy | admin |
| AmazonEKSEditPolicy | 编辑 |
| AmazonEKSViewPolicy | 查看 |

```
$ aws eks list-access-policies
{
    "accessPolicies": [
        {
            "name": "AmazonEKSAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSAdminPolicy"
        },
        {
            "name": "AmazonEKSClusterAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy"
        },
        {
            "name": "AmazonEKSEditPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSEditPolicy"
        },
        {
            "name": "AmazonEKSViewPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSViewPolicy"
        }
    ]
}

$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
```

```
    "accessEntries": []
}
```

在没有集群创建者管理员权限的情况下创建集群时，没有访问条目可用，这是默认情况下创建的唯一条目。

### aw s-auth ConfigMap （已弃用）

Kubernetes 与 AWS 身份验证集成的一种方法是通过 aws-auth ConfigMap，它位于命名空间中。kube-system它负责将 AWS IAM 身份（用户、群组和角色）身份验证映射到 Kubernetes 基于角色的访问控制 (RBAC) 授权。aws-auth ConfigMap 是在您的 Amazon EKS 集群的预配置阶段自动创建的。它最初是为了允许节点加入您的集群而创建的，但如前所述，您也可以使用它 ConfigMap 来添加 RBAC 对 IAM 委托人的访问权限。

要检查您的集群 aws-auth ConfigMap，您可以使用以下命令。

```
kubectl -n kube-system get configmap aws-auth -o yaml
```

这是的默认配置示例aws-authConfigMap。

```
apiVersion: v1
data:
  mapRoles: |
    - groups:
      - system:bootstrappers
      - system:nodes
      - system:node-proxier
      rolearn: arn:aws:iam::<AWS_ACCOUNT_ID>:role/kube-system-<SELF_GENERATED_UUID>
      username: system:node:{{SessionName}}
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-22T18:19:30Z"
  name: aws-auth
  namespace: kube-system
```

它的主会话位于mapRoles区块下方data，它基本上由 3 个参数组成。 ConfigMap

- 群组：要将 IAM 角色映射 group/groups 到的 Kubernetes。这可以是默认群组，也可以是或中指定的自定义群组rolebinding。clusterrolebinding在上面的示例中，我们只声明了系统组。
- rolearn：使用以下格式，将 AWS IAM 角色的 ARN 映射到 group/groups Kubernetes 添加中。arn:<PARTITION>:iam::<AWS_ACCOUNT_ID>:role/role-name

- 用户名：Kubernetes 中要映射到 AWS IAM 角色的用户名。这可以是任何自定义名称。

也可以映射 AWS IAM 用户的权限，data在下面为mapUsers替换 userarn 的 rolearn 参数定义一个新的配置块，但作为最佳实践，始终建议用户改为使用它。aws-auth ConfigMap mapRoles

要管理权限，您可以编辑aws-auth ConfigMap 添加或移除对 Amazon EKS 集群的访问权限。尽管可以aws-auth ConfigMap 手动编辑，但建议使用诸如此类的工具，因为这是一个非常敏感的配置eksctl，不准确的配置可能会将您锁定在 Amazon EKS 集群之外。有关更多详细信息，请查看 ConfigMap下面的 “使用工具对 aws-auth 进行更改” 小节。

### 与 ConfigMap-based 访问管理相比的优势

1. 降低配置错误的风险：直接 API-based 管理可消除与手动 ConfigMap 编辑相关的常见错误。这有助于防止意外删除或语法错误，这些错误可能会将用户锁定在集群之外。
2. 增强的最低权限原则：不再需要集群创建者身份中的集群管理员权限，并允许进行更精细和适当的权限分配。您可以选择为破碎玻璃用例添加此权限。
3. 增强的安全模型：在应用访问条目之前提供对访问条目的内置验证。此外，还提供与 AWS IAM 更紧密的集成以进行身份验证。
4. 简化操作：通过 AWS-native 工具提供更直观的方式来管理权限。

## 集群访问建议

### 将 IAM 身份中心与 CAM API 相结合

- 简化管理：通过将集群访问管理 API 与 IAM Identity Center 配合使用，管理员可以管理 EKS 集群访问权限以及其他 AWS 服务，从而减少了在不同界面之间切换或 ConfigMaps 手动编辑的需求。
- 使用访问条目管理集群外的 IAM 主体的 Kubernetes 权限。您可以使用 EKS API、AWS 命令行界面、AWS 软件开发工具包、AWS 和 AWS 管理控制台来添加和管理对集群的访问权限。CloudFormation这意味着您可以使用与创建集群相同的工具来管理用户。
- 利用本示例中演示的自动化来部署以 AWS IAM 身份中心作为身份提供商、将 CAM API 作为入口点的集群。
- 可以通过访问条目和访问策略将 Kubernetes 用户或群组映射到与 SSO 身份关联的 IAM 委托人，从而应用精细的 Kubernetes 权限。
- 首先，请按照更改身份验证模式使用访问条目，然后按照迁移现有的 aws-auth ConfigMap 条目来访问条目。

## 将 EKS 集群终端节点设为私有

默认情况下，当您配置 EKS 集群时，API 集群终端节点设置为公用，即可以从互联网进行访问。尽管可以从互联网访问，但终端节点仍然被认为是安全的，因为它要求所有 API 请求都必须通过 IAM 进行身份验证，然后由 Kubernetes RBAC 授权。也就是说，如果您的公司安全策略要求您限制从 Internet 访问 API 或禁止您在集群 VPC 之外路由流量，则您可以：

- 将 EKS 集群终端节点配置为私有终端节点。有关此主题的更多信息，请参阅修改集群终端节点访问权限。
- 将集群终端节点保持为公用状态，并指定哪些 CIDR 块可以与集群终端节点通信。这些区块实际上是一组被列入白名单的公有 IP 地址，允许访问集群终端节点。
- 使用一组列入白名单的 CIDR 块配置公共访问并将私有终端节点访问权限设置为已启用。这将允许从特定范围的公有 IP 进行公开访问，同时强制通过 kubelet（工作程序）和 Kubernetes API 之间的所有网络流量通过跨账户 ENI 进行配置，这些跨账户 ENI 在配置控制平面时会配置到集群 VPC 中。

## 请勿使用服务帐号令牌进行身份验证

服务帐号令牌是一种长期存在的静态凭证。如果令牌遭到入侵、丢失或被盗，攻击者可能能够执行与该令牌相关的所有操作，直到服务帐户被删除。有时，对于必须从集群外部使用 Kubernetes API 的应用程序（例如管道应用程序），您可能需要允许例外。 CI/CD如果此类应用程序在 AWS 基础设施（如 EC2 实例）上运行，请考虑使用实例配置文件并将其映射到 Kubernetes RBAC 角色。

## 使用 AWS 资源的最低权限访问权限

IAM 用户无需获得 AWS 资源的权限即可访问 Kubernetes API。如果您需要授予 IAM 用户访问 EKS 集群的权限，请在中`aws-auth` ConfigMap 为该用户创建一个映射到特定 Kubernetes RBAC 组的条目。

## 从集群创建者主体中移除集群管理员权限

默认情况下，Amazon EKS 集群是使用绑定到集群创建者主体的永久`cluster-admin`权限创建的。使用集群访问管理器 API，在使用`API_AND_CONFIG_MAP`或`API`身份验证模式时`false`，无需将此权限设置为，`--access-config bootstrapClusterCreatorAdminPermissions`即可创建集群。撤消此访问权限被认为是最佳实践，可以避免对集群配置进行任何不必要的更改。撤消此访问权限的过程与撤消对集群的任何其他访问权限的过程相同。

API 允许您灵活地仅解除 IAM 委托人与访问策略的关联，在本例中`AmazonEKSClusterAdminPolicy`为。

```
$ aws eks list-associated-access-policies \
    --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN>

$ aws eks disassociate-access-policy --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN. \
    --policy-arn arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy
```

或者完全删除与该cluster-admin权限关联的访问条目。

```
$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
    "accessEntries": []
}

$ aws eks delete-access-entry --cluster-name <CLUSTER_NAME> \
  --principal-arn <IAM_PRINCIPAL_ARN>
```

在事件、紧急情况或无法访问集群的情况下，如果需要，可以再次授予此访问权限。

如果集群仍配置了CONFIG_MAP身份验证方法，则应通过向所有其他用户授予访问集群的权限 aws-auth ConfigMap，配置完成后 aws-auth ConfigMap ，可以删除分配给创建集群的实体的角色，并且只能在发生事件、紧急情况或破碎情形，或者群集损坏而无法访问集群时才可以删除并重新创建。aws-auth ConfigMap 这在生产集群中可能特别有用。

## 当多个用户需要相同的集群访问权限时，请使用 IAM 角色

与其为每个 IAM 用户创建条目，不如允许这些用户担任 IAM 角色并将该角色映射到 Kubernetes RBAC 群组。这将更易于维护，尤其是在需要访问权限的用户数量增加的情况下。

> **Important**
>
> 访问映射为 IAM 实体的 EKS 集群时 aws-auth ConfigMap，所描述的用户名会记录在 Kubernetes 审计日志的用户字段中。如果您使用的是 IAM 角色，则不会记录担任该角色的实际用户，也无法对其进行审计。

如果仍使用 aws-auth ConfigMap 作为身份验证方法，则在向 IAM 角色分配 K8s RBAC 权限时，应在用户名中包含\ {{SessionName}}。这样，审核日志将记录会话名称，这样您就可以跟踪实际用户是谁担任此角色以及日 CloudTrail 志。

```
- rolearn: arn:aws:iam::XXXXXXXXXXXX:role/testRole
  username: testRole:{{SessionName}}
  groups:
    - system:masters
```

> 在 Kubernetes 1.20 及更高版本中，不再需要进行此更改，因为`user.extra.sessionName.0`此更改已添加到 Kubernetes 审计日志中。

## 创建时使用最低特权访问权限 RoleBindings 和 ClusterRoleBindings

就像前面关于授予 AWS 资源访问权限的观点一样，RoleBindings 并且 ClusterRoleBindings 应仅包括执行特定功能所需的权限集。除非绝对必要， ClusterRoles 否则请避免`["*"]`在角色中使用。如果你不确定要分配什么权限，可以考虑使用像 a udit2rbac 这样的工具，根据在 Kubernetes 审计日志中观察到的 API 调用自动生成角色和绑定。

## 使用自动流程创建集群

如前面的步骤所示，在创建 Amazon EKS 集群时，如果不使用使用`API_AND_CONFIG_MAP`或`API`身份验证模式，并且不选择向集群创建者委派`cluster-admin`权限，则在集群的 RBAC 配置中，IAM 实体用户或角色（例如创建集群的联合用户）将自动获得`system:masters`权限。即使是移除此权限的最佳实践，如本文所述，如果使用`CONFIG_MAP`身份验证方法 aws-auth ConfigMap，则无法撤消此访问权限。因此，最好使用与专用 IAM 角色绑定的基础设施自动化管道创建集群，其他用户或实体不具备任何权限，并定期审计该角色的权限、策略以及有权触发管道的人员。此外，不应使用此角色在集群上执行例行操作，而应专门用于由管道触发的集群级别操作，例如通过 SCM 代码更改。

## 使用专用 IAM 角色创建集群

创建 Amazon EKS 集群时，会自动向 IAM 实体用户或角色（例如创建集群的联合用户）授予该集群的 RBAC `system:masters` 配置中的权限。此访问权限无法删除，也不能通过进行管理`aws-auth` ConfigMap。因此，最好创建具有专用 IAM 角色的集群，并定期审计谁可以担任此角色。不应使用此角色在群集上执行例行操作，而是应为此向其他用户授予访问群集`aws-auth` ConfigMap 的权限。配置完成后，应保护该角色，并且仅在临时提升的权限模式下使用 /break glass 以其他方式无法访问集群的场景。`aws-auth` ConfigMap 这在未配置直接用户访问权限的集群中特别有用。

### 定期审核对集群的访问权限

谁需要访问权限可能会随着时间的推移而发生变化。计划定期审计，aws-auth ConfigMap 以查看谁获得了访问权限以及他们被分配了哪些权限。您还可以使用诸如 kubectl-who-can 或 rbac-look up 之类的开源工具来检查绑定到特定服务帐号、用户或群组的角色。当我们进入审计部分时，我们将进一步探讨这个话题。其他想法可以在NCC集团的这篇文章中找到。

### 如果依赖 **aws-auth** 配置地图，请使用工具进行更改

格式不正确的 aws-auth ConfigMap 可能会导致您无法访问集群。如果您需要对进行更改 ConfigMap，请使用工具。

eksctl CL `eksctl` I 包含一个用于向 aws-auth 添加身份映射的命令。 ConfigMap

查看 CLI 帮助：

```
$ eksctl create iamidentitymapping --help
...
```

检查映射到您的 Amazon EKS 集群的身份。

```
$ eksctl get iamidentitymapping --cluster $CLUSTER_NAME --region $AWS_REGION
ARN                                                                    USERNAME
               GROUPS                                                   ACCOUNT
arn:aws:iam::788355785855:role/kube-system-<SELF_GENERATED_UUID>      system:node:
{{SessionName}}     system:bootstrappers,system:nodes,system:node-proxier
```

将 IAM 角色设为集群管理员：

```
$ eksctl create iamidentitymapping --cluster  <CLUSTER_NAME> --region=<region> --arn
 arn:aws:iam::123456:role/testing --group system:masters --username admin
...
```

如需了解更多信息，请查看eksctl文档

keikoproj 的 aws-auth

aws-authby keikoproj 包括一个 cli 和 go 库。

下载并查看帮助 CLI 帮助：

```
$ go get github.com/keikoproj/aws-auth
```

```
...
$ aws-auth help
...
```

或者，aws-auth使用适用于 kubectl 的 ku rew 插件管理器进行安装。

```
$ kubectl krew install aws-auth
...
$ kubectl aws-auth
...
```

请查看 aws-auth 文档 GitHub以获取更多信息，包括 go 库。

AWS IAM Authenticator IAM Auth

该aws-iam-authenticator项目包括用于更新的 CLI ConfigMap。

在上下载发行版 GitHub。

向 IAM 角色添加集群权限：

```
$ ./aws-iam-authenticator add role --rolearn arn:aws:iam::185309785115:role/lil-dev-
role-cluster --username lil-dev-user --groups system:masters --kubeconfig ~/.kube/
config
...
```

### 身份验证和访问管理的替代方法

虽然 IAM 是对需要访问 EKS 集群的用户进行身份验证的首选方式，但也可以使用 OIDC 身份提供商，例如GitHub 使用身份验证代理和 Kubernetes 模拟。AWS 开源博客上已经发布了两个此类解决方案的文章：

- 使用带传送的 GitHub 凭证向 EKS 进行身份验证
- 使用 kube-oidc-proxy 在多个 EKS 集群上进行一致的 OIDC 身份验证

> **Important**
>
> EKS 原生支持 OIDC 身份验证，无需使用代理。如需了解更多信息，请阅读发布博客 Amazon EKS 的 OIDC 身份提供商身份验证简介。有关如何使用 Dex（一种流行的开源 OIDC 提供商，具有用于各种不同身份验证方法的连接器）配置 EKS 的示例，请参阅使用 Dex 和 dex-

k8s-authenticator 向 Amazon EKS 进行身份验证。如博客中所述，经过 OIDC 提供商认证 username/group 的用户将显示在 Kubernetes 审计日志中。

您还可以使用 AWS SSO 将 AW S 与外部身份提供商（例如 Azure AD）联合起来。如果您决定使用此功能，AWS CLI v2.0 包含一个用于创建命名配置文件的选项，可轻松将 SSO 会话与您当前的 CLI 会话关联并担任 IAM 角色。请注意，在运行之前，您必须代入一个角色，`kubectl`因为 IAM 角色用于确定用户的 Kubernetes RBAC 组。

## EKS 容器的身份和凭证

在 Kubernetes 集群中运行的某些应用程序需要获得调用 Kubernetes API 的权限才能正常运行。例如，AWS Load Balancer 控制器需要能够列出服务的终端节点。控制器还需要能够调用 AWS API 来配置和配置 ALB。在本节中，我们将探讨为 Pod 分配权限和权限的最佳实践。

### Kubernetes 服务账号

服务账号是一种特殊类型的对象，允许你为容器分配 Kubernetes RBAC 角色。系统会自动为集群中的每个命名空间创建一个默认服务帐户。当您在不引用特定服务帐户的情况下将 Pod 部署到命名空间时，该命名空间的默认服务帐户将自动分配给该 Pod，而 Secret（即该服务帐户的服务帐户 (JWT) 令牌）将作为卷安装到该 Pod 上。`/var/run/secrets/kubernetes.io/serviceaccount`解码该目录中的服务账号令牌将显示以下元数据：

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-5pv4z",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid":
 "3b36ddb5-438c-11ea-9438-063a49b60fba",
  "sub": "system:serviceaccount:default:default"
}
```

默认服务账号对 Kubernetes API 具有以下权限。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2020-01-30T18:13:25Z"
```

```
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "43"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/system%3Adiscovery
  uid: 350d2ab8-438c-11ea-9438-063a49b60fba
rules:
- nonResourceURLs:
  - /api
  - /api/*
  - /apis
  - /apis/*
  - /healthz
  - /openapi
  - /openapi/*
  - /version
  - /version/
  verbs:
  - get
```

此角色授权未经身份验证和经过身份验证的用户读取 API 信息，并且被认为是安全的，可以公开访问。

当在 Pod 中运行的应用程序调用 Kubernetes API 时，需要为该 Pod 分配一个服务账户，明确授予其调用这些 API 的权限。与用户访问指南类似，角色或 ClusterRole 绑定到服务帐号的权限应仅限于应用程序运行所需的 API 资源和方法，仅限于其他任何内容。要使用非默认服务帐号，只需将 Pod 的`spec.serviceAccountName`字段设置为您要使用的服务帐号名称即可。有关创建服务帐号的更多信息，请参阅 https://kubernetes.io/docs/reference/access-authn-authz/rbac/ #service-account-permissions。

> **Note**
>
> 在 Kubernetes 1.24 之前，Kubernetes 会自动为每个服务账户创建一个密钥。这个秘密安装在/var/run/的吊舱上secrets/kubernetes。 io/serviceaccount 并将被 pod 用来向 Kubernetes API 服务器进行身份验证。在 Kubernetes 1.24 中，服务账号令牌是在容器运行时动态生成的，默认情况下仅在一小时内有效。不会为服务帐号创建密钥。如果您有一个在集群之外运行的应用程序需要向 Kubernetes API 进行身份验证，例如 Jenkins，则需要创建一个类型的密钥`kubernetes.io/service-account-token`以及一个引用服务帐户的注释，例如。`metadata.annotations.kubernetes.io/service-account.name: <SERVICE_ACCOUNT_NAME>`以这种方式创建的密钥不会过期。

## 服务账户的 IAM 角色 (IRSA)

IRSA 是一项允许您为 Kubernetes 服务账户分配 IAM 角色的功能。它通过利用名为服务账户代币量投影的 Kubernetes 功能来工作。当 Pod 配置了引用 IAM 角色的服务账户时，Kubernetes API 服务器将在启动时调用集群的公共 OIDC 发现终端节点。端点对由 Kubernetes 发行的 OIDC 令牌进行加密签名，并将生成的令牌作为一个卷挂载。此签名令牌允许 Pod 调用 AWS API 关联的 IAM 角色。调用 AWS API 时，AWS 开发工具包会调用`sts:AssumeRoleWithWebIdentity`。验证令牌的签名后，IAM 将 Kubernetes 颁发的令牌交换为临时的 AWS 角色证书。

使用 IRSA 时，务必重复使用 AWS 开发工具包会话，以避免不必要地调用 AWS STS。

解码 IRSA 的（JWT）令牌将生成与以下示例类似的输出：

```
{
  "aud": [
    "sts.amazonaws.com"
  ],
  "exp": 1582306514,
  "iat": 1582220114,
  "iss": "https://oidc.eks.us-west-2.amazonaws.com/id/
D43CF17C27A865933144EA99A26FB128",
  "kubernetes.io": {
    "namespace": "default",
    "pod": {
      "name": "alpine-57b5664646-rf966",
      "uid": "5a20f883-5407-11ea-a85c-0e62b7a4a436"
    },
    "serviceaccount": {
      "name": "s3-read-only",
      "uid": "a720ba5c-5406-11ea-9438-063a49b60fba"
    }
  },
  "nbf": 1582220114,
  "sub": "system:serviceaccount:default:s3-read-only"
}
```

此特定令牌通过担任 IAM 角色向 S3 授予 Pod 仅限查看的权限。当应用程序尝试从 S3 读取时，会将该令牌交换为一组临时的 IAM 证书，如下所示：

```
{
    "AssumedRoleUser": {
        "AssumedRoleId": "AROA36C6WWEJULFUYMPB6:abc",
```

```
        "Arn": "arn:aws:sts::123456789012:assumed-role/eksctl-winterfell-addon-
iamserviceaccount-de-Role1-1D61LT75JH3MB/abc"
    },
    "Audience": "sts.amazonaws.com",
    "Provider": "arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-
west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128",
    "SubjectFromWebIdentityToken": "system:serviceaccount:default:s3-read-only",
    "Credentials": {
        "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        "SessionToken": "FwoGZXIvYXdzEGMaDMLxAZkuLpmSwYXShiL9A1S0X87VBC1mHCrRe/
pB2oesl1eXxUYnPJyC9ayOoXMvqXQsomq0xs6OqZ3vaa5Iw1HIyA4Cv1suLaOCoU3hNvOIJ6C94H1vU0siQYk7DIq9Av5RZ
Zv2A6zp5xGz9cWj2f0aD9v66vX4bexOs5t/YYhwuwAvkkJPSIGvxja0xRThnceHyFHKtj0Hbi/
PWAtlI8YJcDX69cM30JAHDdQHltm/4scFptW1hlvMaPWReCAaCrsHrATyka7ttw5YlUyvZ8EPogj6fwHlxmrXM9h1Bqdiko
LgkWsCTGlYcY8z3zkigJMbYn07ewTL5Ss7LazTJJa758I7PZan/
v3xQHd5DEc5WBneiV3iOznDFgup0VAMkIviVjVCkszaPSVEdK2NU7jtrh6Jfm7bU/3P6ZGCkyDLIa8MBn9KPXeJd/
yjTk5IifIwO/mDpGNUribg6TPxhzZ8b/XdZO1kS1gVgqjXyVCM+BRBh6C4H21w/eMzjCtDIpoxt5rGKL6Nu/
IFMipoC4fgx6LIIHwtGYMG7SWQi7OsMAkiwZRg0n68/RqWgLzBt/4pfjSRYuk=",
        "Expiration": "2020-02-20T18:49:50Z",
        "AccessKeyId": "ASIAIOSFODNN7EXAMPLE"
    }
}
```

作为 EKS 控制平面一部分运行的变异 webhook 将 AWS 角色 ARN 和网络身份令牌文件的路径作为环境变量注入 Pod。这些值也可以手动提供。

```
AWS_ROLE_ARN=arn:aws:iam::AWS_ACCOUNT_ID:role/IAM_ROLE_NAME
AWS_WEB_IDENTITY_TOKEN_FILE=/var/run/secrets/eks.amazonaws.com/serviceaccount/token
```

当预计代币超过其总TTL的80%或24小时后，kubelet将自动轮换该代币。AWS 软件开发工具包负责在令牌轮换时重新加载令牌。有关 IRSA 的更多信息，请参阅 https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-technical-overview.html.

## EKS 容器组身份

EKS Pod Identities 是 re: Invent 2023 上推出的一项功能，允许您为 kubernetes 服务账户分配 IAM 角色，而无需为 AWS 账户中的每个集群配置 Open Id Connect (OIDC) 身份提供商 (IDP)。要使用 EKS Pod Identity，您必须在每个符合条件的工作节点上部署一个作为 DaemonSet Pod 运行的代理。此代理作为 EKS 提供给您， Add-on 并且是使用 EKS Pod 身份功能的先决条件。您的应用程序必须使用支持版本的 AWS 开发工具包才能使用此功能。

为 Pod 配置 EKS Pod 身份后，EKS 将在上安装并刷新容器身份令牌/var/run/secrets/pods.eks.amazonaws.com/serviceaccount/eks-pod-identity-token。AWS 开发工具

包将使用此令牌与 EKS Pod Identity Agent 通信，后者使用 pod 身份令牌和代理的 IAM 角色通过调用 AssumeRoleForPodIdentityAPI 为您的容器创建临时证书。交付给您的 pod 的 pod 身份令牌是从您的 EKS 集群颁发的 JWT，并经过加密签名，并带有相应的 JWT 声明，可与 EKS Pod 身份一起使用。

要了解有关 EKS Pod 身份的更多信息，请参阅此博客。

您无需对应用程序代码进行任何修改即可使用 EKS Pod 身份。支持的 AWS 开发工具包版本将使用凭证提供者链自动发现通过 EKS Pod 身份提供的证书。与 IRSA 一样，EKS Pod 身份会在您的 pod 中设置变量，以指导他们如何查找 AWS 证书。

使用 EKS Pod 身份的 IAM 角色

- EKS Pod 身份只能直接担任与 EKS 集群属于同一 AWS 账户的 IAM 角色。要访问其他 AWS 账户中的 IAM 角色，您必须通过在软件开发工具包配置或应用程序代码中配置配置文件来代入该角色。
- 为服务账户配置 EKS Pod 身份时，配置 Pod 身份关联的人员或进程必须拥有该角色的`iam:PassRole`权限。
- 每个服务账户只能通过 EKS Pod Identities 关联一个 IAM 角色，但是您可以将相同的 IAM 角色与多个服务账户相关联。
- 与 EKS Pod 身份一起使用的 IAM 角色必须允许`pods.eks.amazonaws.com`服务主体代入这些角色并设置会话标签。以下是允许 EKS Pod Identities 使用 IAM 角色的角色信任策略示例：

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "pods.eks.amazonaws.com"
      },
      "Action": [
        "sts:AssumeRole",
        "sts:TagSession"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceOrgId": "${aws:ResourceOrgId}"
        }
      }
    }
  ]
```

```
}
```

AWS 建议使用诸如此类的条件键`aws:SourceOrgId`来帮助防范跨服务混乱的副手问题。在上面的角色信任策略示例中，`ResourceOrgId`是一个变量，等于 AWS 账户所属的 AWS 组织的 AWS 组织标识。EKS 将传入一个与担任 EKS Pod Identitions 角色时的值`aws:SourceOrgId`相等的值。

ABAC 和 EKS Pod 身份

当 EKS Pod 身份担任 IAM 角色时，它会设置以下会话标签：

| EKS Pod 身份会话标签 | 值 |
| --- | --- |
| kubernetes 命名空间 | 与 EKS Pod 身份关联的 pod 运行的命名空间。 |
| kubernetes 服务账号 | 与 EKS Pod 身份关联的 kubernetes 服务账户的名称 |
| eks-cluster-arn | EKS 集群的 ARN，例如 `arn:${Partition}:eks:${Region}:${Account}:cluster/${ClusterName}` 集群 ARN 是唯一的，但是如果在同一区域、同一 AWS 账户中删除并重新创建具有相同名称的集群，则该集群将具有相同的 ARN。 |
| eks 集群名称 | EKS 集群的名称。请注意，您的 AWS 账户中的 EKS 集群名称可以相同，其他 AWS 账户中的 EKS 集群名称可以相同。 |
| kubernetes-pod-name | EKS 中吊舱的名称。 |
| kubernetes-pod-uid | EKS 中吊舱的 UID。 |

这些会话标签允许您使用基于属性的访问控制 (ABAC)，仅向特定的 kubernetes 服务账户授予对您的 AWS 资源的访问权限。这样做时，必须明白 kubernetes 服务帐户仅在命名空间中是唯一的，而 kubernetes 命名空间仅在 EKS 集群中是唯一的。可以使用`aws:PrincipalTag/<tag-key>`全局条件密钥在 AWS 策略中访问这些会话标签，例如 `aws:PrincipalTag/eks-cluster-arn`

例如，如果您只想向特定的服务账户授予访问权限，以便通过 IAM 或资源策略访问您账户中的 AWS 资源，则需要检查`eks-cluster-arn`并`kubernetes-namespace`标记，并确保只有目

标集群中的服务账户才能访问该资源，因为其他集群可能具有相同的kubernetes-service-accounts和kubernetes-namespaces。kubernetes-service-account

此示例 S3 存储桶策略仅在eks-cluster-arn所有对象都满足预期值时才授予访问其所连接的 S3 存储桶中的对象的权限，其中 EKS 集群托管在 AWS 账户中111122223333。kubernetes-service-account kubernetes-namespace

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "s3:*",
            "Resource":            [
                "arn:aws:s3:::ExampleBucket/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalTag/kubernetes-service-account": "s3objectservice",
                    "aws:PrincipalTag/eks-cluster-arn": "arn:aws:eks:us-
west-2:111122223333:cluster/ProductionCluster",
                    "aws:PrincipalTag/kubernetes-namespace": "s3datanamespace"
                }
            }
        }
    ]
}
```

## EKS Pod 身份与 IRSA 的比较

EKS Pod 身份和 IRSA 都是向 EKS 容器提供临时 AWS 证书的首选方式。除非您有 IRSA 的特定用例，否则我们建议您在使用 EKS 时使用 EKS Pod 身份。此表有助于比较这两个功能。

| # | EKS 容器组身份 | IRSA |
| --- | --- | --- |
| 需要权限才能在您的 AWS 账户中创建 OIDC IDP？ | 否 | 是 |

| # | EKS 容器组身份 | IRSA |
|---|---|---|
| 每个集群需要唯一的 IDP 设置 | 否 | 是 |
| 设置与 ABAC 一起使用的相关会话标签 | 是 | 否 |
| 需要一个 iam：PassRole 支票？ | 是 | 否 |
| 使用您的 AWS 账户中的 AWS STS 配额？ | 否 | 是 |
| 可以访问其他 AWS 账户 | 间接使用角色链 | 直接使用 sts：AssumeRoleWithWebIdentity |
| 与 AWS 软件开发工具包兼容 | 支持 | 是 |
| 需要在节点上安装 Pod 身份代理守护程序吗？ | 是 | 否 |

# EKS pod 的身份和凭证推荐

## 更新 aws-node 守护程序集以使用 IRSA

目前，aws-node 守护程序集已配置为使用分配给 EC2 实例的角色为 pod 分配 IP。该角色包括多个 AWS 托管策略，例如 Amazoneks_cni_policy，这些策略实际上允许在节点上运行的所有 pod 访问 ENI EC2ContainerRegistryReadOnly 、IP 地址或从 ECR 提取映像。 attach/detach assign/unassign 由于这会给您的集群带来风险，因此建议您更新 aws-node 守护程序集以使用 IRSA。可以在本指南的存储库中找到执行此操作的脚本。

aws-node 守护程序集支持 v1.15.5 及更高版本中的 EKS Pod 身份。

## 限制访问分配给工作节点的实例配置文件

当你使用 IRSA 或 EKS Pod Identitions 时，它会先更新 Pod 的凭证链以使用 IRSA 或 EKS Pod 身份，但是，Pod 仍然可以继承分配给工作节点的实例配置文件的权限。对于不需要这些权限的 Pod，您可以阻止对实例元数据的访问，以帮助确保您的应用程序仅拥有所需的权限，而不是其节点。

> **Warning**
>
> 阻止对实例元数据的访问将阻止不使用 IRSA 或 EKS Pod 身份的 Pod 继承分配给工作节点的角色。

您可以通过要求实例仅使用 imdsv2 并将跳数更新为 1 来阻止对实例元数据的访问，如下例所示。您也可以将这些设置包含在节点组的启动模板中。请勿禁用实例元数据，因为这将阻止诸如节点终止处理程序之类的组件和其他依赖实例元数据的组件正常运行。

```
$ aws ec2 modify-instance-metadata-options --instance-id <value> --http-tokens required
 --http-put-response-hop-limit 1
...
```

如果您使用 Terraform 创建用于托管节点组的启动模板，请添加元数据块以配置跳数，如以下代码片段所示：

```
tf hl_lines="7" resource "aws_launch_template" "foo" { name = "foo" …
 metadata_options { http_endpoint = "enabled" http_tokens = "required"
http_put_response_hop_limit = 1 instance_metadata_tags = "enabled" } …
```

您还可以通过操作节点上的 iptables 来阻止容器对 EC2 元数据的访问。有关此方法的更多信息，请参阅限制对实例元数据服务的访问。

如果您的应用程序使用的是不支持 IRSA 或 EKS Pod 身份的旧版 AWS 开发工具包，则应更新软件开发工具包版本。

## 将 IRSA 角色的 IAM 角色信任策略范围限定为服务账户名称、命名空间和集群

信任策略的范围可以限定为命名空间或命名空间中的特定服务帐号。使用 IRSA 时，最好通过包含服务帐户名称来使角色信任策略尽可能明确。这将有效地防止同一个命名空间中的其他 Pod 担任该角色。当您使用 CLI `eksctl` 创建服务 accounts/IAM 角色时，它会自动执行此操作。有关更多信息，请参阅 https://eksctl.io/usage/iamserviceaccounts/。

直接使用 IAM 时，这就是在角色的信任策略中添加条件，该策略使用条件来确保`:sub`声明是您期望的命名空间和服务账号。举个例子，之前我们有一个 IRSA 代币，其子声明为 “system: serviceaccount: default: s3-只读”。这是`default`命名空间，服务帐号是`s3-read-only`。你可以使用如下条件来确保只有集群中给定命名空间中的服务账号才能担任该角色：

```
 "Condition": {
     "StringEquals": {
         "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:aud":
"sts.amazonaws.com",
         "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:sub":
"system:serviceaccount:default:s3-read-only"
     }
 }
```

### 每个应用程序使用一个 IAM 角色

对于 IRSA 和 EKS Pod Identity，最佳做法是为每个应用程序分配自己的 IAM 角色。这提高了隔离性，因为您可以在不影响另一个应用程序的情况下修改一个应用程序，并且允许您通过仅向应用程序授予所需的权限来应用权限最小的主体。

将 ABAC 与 EKS Pod Identity 配合使用时，您可以在多个服务账户中使用通用 IAM 角色，并依靠其会话属性进行访问控制。这在大规模运营时特别有用，因为 ABAC 允许您使用更少的 IAM 角色进行操作。

### 当您的应用程序需要访问 IMDS 时，请使用 imdsv2 并将 EC2 实例的跳跃限制增加到 2

IMDSv2 要求您使用 PUT 请求来获取会话令牌。初始 PUT 请求必须包含会话令牌的 TTL。较新版本的 AWS SDK 将自动处理此问题以及上述令牌的续订。同样重要的是要注意，EC2 实例的默认跳数限制被故意设置为 1，以防止 IP 转发。因此，请求在 EC2 实例上运行的会话令牌的 Pod 最终可能会超时并回退到使用 imdsv1 数据流。EKS 通过同时启用 v1 和 v2 并将由 eksctl 或使用官方模板配置的节点上的跳跃限制更改为 2 来增加对 imdsv2 的支持。 CloudFormation

### 禁用自动挂载服务账号令牌

如果您的应用程序不需要调用 Kubernetes API，请将应用程序的`automountServiceAccountToken`属性设置为，或者`false`在每个命名空间中修补默认服务帐户，使其不再自动挂载到 pod 上。 PodSpec例如：

```
kubectl patch serviceaccount default -p $'automountServiceAccountToken: false'
```

### 为每个应用程序使用专用服务帐户

每个应用程序都应有自己的专用服务帐户。这适用于 Kubernetes API 的服务账户以及 IRSA 和 EKS Pod Identity。

> **Important**
>
> 如果您在使用 IRSA 时采用集群升级 blue/green 方法而不是执行就地集群升级，则需要使用新集群的 OIDC 终端节点更新每个 IRSA IAM 角色的信任策略。 blue/green 集群升级是指创建一个在旧集群旁边运行较新版本的 Kubernetes 的集群，并使用负载均衡器或服务网格将流量从在旧集群上运行的服务无缝转移到新集群。在 EKS Pod Identity 中使用 blue/green 集群升级时，您将在新集群中的 IAM 角色和服务账户之间创建 Pod 身份关联。如果您有`sourceArn`条件，请更新 IAM 角色信任策略。

## 以非 root 用户身份运行应用程序

默认情况下，容器以 root 用户身份运行。虽然这允许他们读取 Web 身份令牌文件，但以 root 用户身份运行容器并不是最佳实践。或者，可以考虑将`spec.securityContext.runAsUser`属性添加到 PodSpec。的值`runAsUser`是任意值。

在以下示例中，Pod 中的所有进程都将在`runAsUser`字段中指定的用户 ID 下运行。

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
```

当你以非 root 用户身份运行容器时，它会阻止容器读取 IRSA 服务账户令牌，因为默认情况下，该令牌被分配了 0600 [root] 权限。如果你更新容器的 SecurityContext 以包含 fsgroup=65534 [Nobody]，它将允许容器读取令牌。

```
spec:
  securityContext:
    fsGroup: 65534
```

在 Kubernetes 1.19 及更高版本中，不再需要进行此更改，应用程序无需将其添加到 Nobody 组即可读取 IRSA 服务账户令牌。

### 授予对应用程序的最低特权访问权限

Acti@@ on Hero 是一个实用工具，您可以与应用程序一起运行，以识别应用程序正常运行所需的 AWS API 调用和相应的 IAM 权限。它与 IAM Access Advisor 类似，因为它可以帮助您逐步限制分配给应用程序的 IAM 角色的范围。有关更多信息，请参阅有关授予对 AWS 资源的最低权限访问权限的文档。

考虑为与 IRSA 和 Pod 身份一起使用的 IAM 角色设置权限边界。您可以使用权限边界来确保 IRSA 或 Pod Identities 使用的角色不能超过最大权限级别。有关权限边界入门的示例指南以及权限边界策略示例，请参阅此 github 存储库。

### 查看并撤消对您的 EKS 集群的不必要的匿名访问权限

理想情况下，应禁用所有 API 操作的匿名访问权限。匿名访问权限是通过为 Kubernetes 内置用户 ClusterRoleBinding 系统创建 RoleBinding 或来授予的：匿名。你可以使用 rbac-lookup 工具来识别 system: anonymous 用户对你的集群拥有的权限：

```
./rbac-lookup | grep -P 'system:(anonymous)|(unauthenticated)'
system:anonymous               cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:public-info-
viewer
```

ClusterRole 除了 system: public-info-viewer 以外的任何角色都不应绑定到系统:匿名用户或系统:未经身份验证的群组。

在特定 API 上启用匿名访问可能有一些正当理由。如果您的集群是这种情况，请确保匿名用户只能访问这些特定的 API，并且在未经身份验证的情况下公开这些 API 不会使您的集群容易受到攻击。

在 Kubernetes/EKS 版本 1.14 之前，默认情况下，system: 未经身份验证的组与 system: discovery 和 system: basic-user 相关联。 ClusterRoles 请注意，即使您已将集群更新到 1.14 或更高版本，这些权限仍可能在您的集群上启用，因为集群更新不会撤消这些权限。要检查除了 system: public-info-viewer 之外哪些ClusterRoles 有 “system: unautify”，你可以运行以下命令（需要 jq util）：

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer")
 | .metadata.name'
```

而且，可以使用以下方法从除 “system: public-info-viewer” 之外的所有角色中删除 “system: unauthentify

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer") |
 del(.subjects[] | select(.name =="system:unauthenticated"))' | kubectl apply -f -
```

或者，你可以通过 kubectl describe 和 kubectl edit 手动检查和删除它。要检查 system: 未经身份验证的组是否在您的集群上拥有 system: discovery 权限，请运行以下命令：

```
kubectl describe clusterrolebindings system:discovery

Name:         system:discovery
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:discovery
Subjects:
  Kind   Name                   Namespace
  ----   ----                   ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

要检查 system: 未经身份验证的组在您的集群上是否拥有 system: basic-user 权限，请运行以下命令：

```
kubectl describe clusterrolebindings system:basic-user

Name:         system:basic-user
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:basic-user
Subjects:
  Kind   Name                   Namespace
  ----   ----                   ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

如果 system: 未经身份验证的组绑定到集群 ClusterRoles 上的 system: discovery system and/or: basic-user，则应取消这些角色与 system: 未经身份验证的组的关联。ClusterRoleBinding 使用以下命令编辑 system: discovery：

```
kubectl edit clusterrolebindings system:discovery
```

上面的命令将在编辑器中打开 system: discovery ClusterRoleBinding 的当前定义，如下所示：

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will
 be
# reopened with the relevant failures.
#
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2021-06-17T20:50:49Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "24502985"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Adiscovery
  uid: b7936268-5043-431a-a0e1-171a423abeb6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:discovery
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:unauthenticated
```

从上述编辑器屏幕的 “主题” 部分中删除 system: 未经身份验证的群组的条目。

对 system: ClusterRoleBinding basic-user 重复相同的步骤。

## 在 IRSA 中重复使用 AWS 开发工具包会话

当您使用 IRSA 时，使用 AWS 开发工具包编写的应用程序会使用传送到您的 pod 的令牌`sts:AssumeRoleWithWebIdentity`来调用生成临时 AWS 证书。这与其他 AWS 计算服务不同，在这些服务中，计算服务直接向 AWS 计算资源（例如 lambda 函数）提供临时 AWS 证书。这意味着，每次初始化 AWS 开发工具包会话时，都会调用 AWS STS f `AssumeRoleWithWebIdentity` or。如果您的应用程序快速扩展并初始化了许多 AWS SDK 会话，则可能会遇到来自 AWS STS 的限制，因为您的代码会发出很多调用。`AssumeRoleWithWebIdentity`

为避免出现这种情况，我们建议您在应用程序中重复使用 AWS 开发工具包会话，这样就不会对进行`AssumeRoleWithWebIdentity`不必要的调用。

在以下示例代码中，使用 boto3 python SDK 创建了一个会话，该会话用于创建客户端并与 Amazon S3 和 Amazon SQS 进行交互。 `AssumeRoleWithWebIdentity`仅调用一次，AWS 开发工具包将自动刷新证书到期`my_session`时的证书。

```
import boto3

= Create your own session

my_session = boto3.session.Session()

= Now we can create low-level clients from our session

sqs = my_session.client(``sqs``) s3 = my_session.client(``s3``)

s3response = s3.list_buckets() sqsresponse = sqs.list_queues()

#print the response from the S3 and SQS APIs print("`s3 response:`")
print(s3response) print("`—`") print("`sqs response:`")
print(sqsresponse)
```

如果您要使用 IRSA 将应用程序从其他 AWS 计算服务（例如 EC2）迁移到 EKS，则这是一个特别重要的细节。在其他计算服务上，除非您指示，否则初始化 AWS 开发工具包会话不会调用 AWS STS。

## 替代方法

虽然 IRSA 和 EKS Pod 身份是向容器分配 AWS 身份的首选方式，但它们要求您在应用程序中包含最新版本的 AWS 软件开发工具包。有关当前支持 IRSA 的软件开发工具包的完整列表，请参阅https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-minimum-sdk.html，有关 EKS Pod 身份，请参阅https://docs.aws.amazon.com/eks/latest/userguide/pod-id-minimum-sdk.html.

如果您的应用程序无法使用兼容的 SDK 立即更新，则有几种社区构建的解决方案可用于向 Kubernetes pod 分配 IAM 角色，包括 kube2iam 和 kiam。尽管 AWS 不认可、宽恕也不支持这些解决方案的使用，但整个社区经常使用它们来实现与 IRSA 和 EKS Pod Identities 相似的结果。

如果您需要使用这些非 AWS 提供的解决方案，请进行尽职调查，并确保您了解这样做可能带来的安全影响。

### 工具和资源

- Amazon EKS 安全沉浸式研讨会——Identity and Access 管理
- Terraform EKS 蓝图模式——完全私有的亚马逊 EKS 集群
- Terraform EKS 蓝图模式——适用于 Sign-On 亚马逊 EKS 集群的 IAM 身份中心单一身份中心
- Terraform EKS 蓝图模式——适用于 Sign-On 亚马逊 EKS 集群的 Okta Single
- audit2rbac
- rbac.dev Kubernetes RBAC 的其他资源清单，包括博客和工具
- 动作英雄
- kube2iam
- kiam

## Pod 安全

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Pod 规范包括各种不同的属性，这些属性可以增强或削弱您的整体安全状况。作为 Kubernetes 从业者，你最关心的问题应该是防止在容器中运行的进程逃离容器运行时的隔离边界并获得对底层主机的访问权限。

### Linux 功能

默认情况下，在容器内运行的进程在 [Linux] root 用户的上下文中运行。尽管 root 在容器中的操作受到容器运行时分配给容器的一组 Linux 功能的部分限制，但这些默认权限可能允许攻击者升级其权限，and/or访问绑定到主机的敏感信息，包括 Secrets 和。 ConfigMaps以下是分配给容器的默认功能列表。有关每项权能的其他信息，请参阅 http://man7.org/linux/man-pages/man7/capabilities.7.html.

`CAP_AUDIT_WRITE, CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID, CAP_KILL, CAP_MKNOD, CAP_NET_BIND_SERVICE, CAP_NET_RAW, CAP_SETGID, CAP_SETUID, CAP_SETFCAP, CAP_SETPCAP, CAP_SYS_CHROOT`

Example

默认情况下，会为 EC2 和 Fargate 容器分配上述功能。此外，只能从 Fargate 吊舱中删除 Linux 功能。

以特权身份运行的 Pod 会继承主机上与 root 相关的所有 Linux 功能。如果可能，应避免这种情况。

## 节点授权

所有 Kubernetes 工作节点都使用一种名为 “节点授权” 的授权模式。节点授权授权所有源自 kubelet 的 API 请求，并允许节点执行以下操作：

读取操作：

- 务
- 端点
- 节点
- 容器组（pod）
- 与绑定到 kubelet 节点的 pod 相关的机密、配置映射、永久卷声明和永久卷

写入操作：

- 节点和节点状态（启用`NodeRestriction`准入插件以限制 kubelet 修改自己的节点）
- pod 和 pod 状态（启用`NodeRestriction`准入插件以限制 kubelet 修改绑定到自身的 pod）
- events

Auth-related 操作：

- Read/write 访问用于 TLS 引导的 CertificateSigningRequest (CSR) API
- 能够创建 TokenReview 和 authentication/authorization 执行委托 SubjectAccessReview 支票

EKS 使用节点限制准入控制器，该控制器仅允许节点修改绑定到该节点的一组有限的节点属性和容器对象。尽管如此，设法访问主机的攻击者仍然能够从 Kubernetes API 中收集有关环境的敏感信息，从而允许他们在集群内横向移动。

# Pod 安全解决方案

## Pod 安全政策 (PSP)

过去，Pod 安全策略 (PSP) 资源用于指定 Pod 在创建 Pod 之前必须满足的一组要求。从 Kubernetes 1.21 版本开始，PSP 已被弃用。它们计划在 Kubernetes 1.25 版本中移除。

> **Important**
>
> 在 Kubernet@@ es 版本 1.21 中，PSP 已被弃用。在版本 1.25 或大约 2 年之前，你需要过渡到替代版本。本文档解释了此次弃用的动机。

## 迁移到新的 pod 安全解决方案

由于从 Kubernetes v1.25 起，PSP 已被移除，因此集群管理员和操作员必须更换这些安全控件。有两种解决方案可以满足这一需求：

- Policy-as-code 来自 Kubernetes 生态系统的 (PAC) 解决方案
- Kubernetes Pod 安全标准 (PSS)

PAC 和 PSS 解决方案都可以与 PSP 共存；它们可以在移除 PSP 之前在集群中使用。这样可以简化从 PSP 迁移时的采用。在考虑从 PSP 迁移到 PSS 时，请参阅此文档。

Kyverno是下文概述的PAC解决方案之一，在从PSP迁移到其解决方案时，博客文章中概述了具体的指导，包括类似的策略、功能比较和迁移程序。有关 Pod 安全准入 (PSA) 的迁移到 Kyverno 的其他信息和指南已在此处的 AWS 博客上发布。

## Policy-as-code (PAC)

Policy-as-code (PAC) 解决方案提供护栏，通过规定的自动控制来引导集群用户，防止不良行为。PAC 使用 Kubernetes 动态准入控制器通过 webhook 调用拦截 Kubernetes API 服务器请求流，并根据以代码形式编写和存储的策略对请求负载进行变异和验证。在 API 服务器请求导致集群发生变化之前，就会发生变更和验证。PAC 解决方案使用策略根据分类法和值对 API 服务器请求有效载荷进行匹配和操作。

Kubernetes 有几种开源 PAC 解决方案可供选择。这些解决方案不是 Kubernetes 项目的一部分；它们来自 Kubernetes 生态系统。下面列出了一些 PAC 解决方案。

- OPA/Gatekeeper
- 开放策略代理 (OPA)
- Kyverno
- Kubewarden
- jsPolicy

有关 PAC 解决方案以及如何帮助您选择适合您需求的解决方案的更多信息，请参阅以下链接。

- Policy-basedKubernetes 的对策 — 第 1 部分
- Policy-basedKubernetes 的对策 — 第 2 部分

## 吊舱安全标准 (PSS) 和吊舱安全准入 (PSA)

为了回应 PSP 的弃用以及对开箱即用控制容器安全的持续需求，Kubernetes Auth 特别兴趣小组使用内置的 Kubernetes 解决方案创建了 Pod 安全标准 (PSS) 和 Pod 安全准入 (PSA)。PSA 的工作包括一个准入控制器 webhook 项目，该项目实现了 PSS 中定义的控制措施。这种准入控制器方法类似于 PAC 解决方案中使用的方法。

根据Kubernetes的文档，PSS “`定义了三种不同的策略来广泛涵盖安全领域。这些策略是累积性的，范围从高度允许到高度限制。`”

这些策略定义为：

- 特权：不受限制（不安全）的策略，提供尽可能广泛的权限级别。此策略允许已知的权限升级。这是政策的缺失。这对于日志代理、CNI、存储驱动程序和其他需要特权访问权限的系统范围的应用程序非常有用。
- 基准：限制性最低的策略，可防止已知的权限升级。允许默认（最低限度指定）Pod 配置。基准政策禁止使用 HostNetwork、HostPid、HostPid、HostPort、HostPort、无法添加 Linux 功能以及其他一些限制。
- 限制：严格限制政策，遵循当前 Pod 强化最佳实践。此策略继承了基准，并增加了更多限制，例如无法以 root 用户或根组的身份运行。受限的策略可能会影响应用程序的运行能力。它们主要针对运行安全关键型应用程序。

这些策略定义了 Pod 执行的配置文件，分为特权访问与受限访问的三个级别。

为了实现PSS定义的控制措施，PSA在三种模式下运行：

- 强制执行：违反策略将导致 pod 被拒绝。
- 审计：违反策略将触发在审计日志中记录的事件中添加审计注释，但除此之外是允许的。
- 警告：违反策略将触发面向用户的警告，但除此之外是允许的。

这些模式和配置文件（限制）级别是在 Kubernetes 命名空间级别使用标签配置的，如下例所示。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: restricted
```

单独使用时，这些操作模式会有不同的响应，从而产生不同的用户体验。如果相应的 PodSpecs 违反了配置的限制级别，则强制模式将阻止创建 Pod。但是，在这种模式下，即使其中的 PodSpec 违反了所应用的 PSS，也不会阻止创建容器的非 Pod Kubernetes 对象（例如 Deploymentes）应用于集群。在这种情况下，将应用部署，而将阻止应用 pod。

这是一种困难的用户体验，因为没有立即迹象表明成功应用的 Deployment 对象掩盖了 pod 创建失败。违规的 PodSpecs 不会创建 pod。使用检查部署资源`kubectl get deploy <DEPLOYMENT_NAME> -oyaml`将显示来自失败的 pod `.status.conditions` 元素的消息，如下所示。

```
...
status:
  conditions:
    - lastTransitionTime: "2022-01-20T01:02:08Z"
      lastUpdateTime: "2022-01-20T01:02:08Z"
      message: 'pods "test-688f68dc87-tw587" is forbidden: violates PodSecurity
 "restricted:latest":
        allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false),
        unrestricted capabilities (container "test" must set
 securityContext.capabilities.drop=["ALL"]),
        runAsNonRoot != true (pod or container "test" must set
 securityContext.runAsNonRoot=true),
        seccompProfile (pod or container "test" must set
 securityContext.seccompProfile.type
        to "RuntimeDefault" or "Localhost")'
      reason: FailedCreate
```

```
      status: "True"
      type: ReplicaFailure
...
```

在审计和警告模式下，Pod 限制并不能阻止创建和启动违规的 Pod。但是，在这些模式下，当 Pod 以及创建 Pod 的对象包含违规的 PodSp ecs 时，会分别触发 API 服务器审核日志事件的审计注释和向 API 服务器客户端（例如 kubectl）发出的警告。kubectl警告消息如下所示。

```
Warning: would violate PodSecurity "restricted:latest":
 allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container
 "test" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod
 or container "test" must set securityContext.runAsNonRoot=true), seccompProfile (pod
 or container "test" must set securityContext.seccompProfile.type to "RuntimeDefault"
 or "Localhost")
deployment.apps/test created
```

PSA 审计和警告模式在引入 PSS 时非常有用，不会对群集操作产生负面影响。

PSA 的操作模式并不相互排斥，可以累积使用。如下所示，可以在单个命名空间中配置多种模式。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

在上面的示例中，在应用 Deployments 时提供了用户友好的警告和审计注释，同时还在 pod 级别提供了违规的强制执行。实际上，多个 PSA 标签可以使用不同的配置文件级别，如下所示。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: restricted
```

在上面的示例中，PSA 配置为允许创建满足基准配置文件级别的所有容器，然后对违反受限配置文件级别的 pod（以及创建 Pod 的对象）发出警告。这是一种有用的方法，可以确定从基线更改为受限配置文件时可能产生的影响。

现有 Pod

如果将包含现有 pod 的命名空间修改为使用限制性更强的 PSS 配置文件，则审计和警告模式将生成相应的消息；但是，强制模式不会删除 pod。警告消息如下所示。

```
Warning: existing pods in namespace "policy-test" violate the new PodSecurity enforce
 level "restricted:latest"
Warning: test-688f68dc87-htm8x: allowPrivilegeEscalation != false, unrestricted
 capabilities, runAsNonRoot != true, seccompProfile
namespace/policy-test configured
```

豁免

PSA 使用豁免来排除对原本适用的 pod 的违规行为的强制执行。这些豁免如下所列。

- 用户名：将忽略使用免除身份验证（或模仿）用户名的用户发出的请求。
- RuntimeClassNames: 指定豁免运行时类名称的 pod 和工作负载资源将被忽略。
- 命名空间：豁免命名空间中的 Pod 和工作负载资源将被忽略。

作为 API 服务器配置的一部分，这些豁免在 PSA 准入控制器配置中静态应用。

在验证 Webhook 实现中，可以在 Kubernetes ConfigMap资源中配置豁免，该资源作为卷挂载到 pod-security webhook 容器中。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pod-security-webhook
  namespace: pod-security-webhook
data:
  podsecurityconfiguration.yaml: |
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "restricted"
      enforce-version: "latest"
```

```
      audit: "restricted"
      audit-version: "latest"
      warn: "restricted"
      warn-version: "latest"
    exemptions:
      # Array of authenticated usernames to exempt.
      usernames: []
      # Array of runtime class names to exempt.
      runtimeClasses: []
      # Array of namespaces to exempt.
      namespaces: ["kube-system","policy-test1"]
```

如上面的 ConfigMap YAML 所示，集群范围的默认 PSS 级别已设置为限制所有 PSA 模式，包括审计、强制和警告。这会影响所有命名空间，但豁免的命名空间除外:. `namespaces: ["kube-system","policy-test1"]` 此外，在ValidatingWebhookConfiguration资源中，如下所示，pod-security-webhook 命名空间也被排除在配置的 PSS 之外。

```
...
webhooks:
  # Audit annotations will be prefixed with this name
  - name: "pod-security-webhook.kubernetes.io"
    # Fail-closed admission webhooks can present operational challenges.
    # You may want to consider using a failure policy of Ignore, but should
    # consider the security tradeoffs.
    failurePolicy: Fail
    namespaceSelector:
      # Exempt the webhook itself to avoid a circular dependency.
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: NotIn
          values: ["pod-security-webhook"]
...
```

> **Important**
>
> 在 Kubernetes v1.25 中，Pod Security 招生已升级到稳定版。如果您想在默认启用 Pod 安全准入功能之前使用该功能，则需要安装动态准入控制器（改变 webhook）。可以在此处找到安装和配置 webhook 的说明。

## 在策略即代码和 Pod 安全标准之间进行选择

Pod 安全标准 (PSS) 的开发是为了取代 Pod 安全政策 (PSP)，它提供了一种内置于 Kubernetes 的解决方案，不需要来自 Kubernetes 生态系统的解决方案。话虽如此，政策即代码 (PAC) 解决方案要灵活得多。

以下优点和缺点列表旨在帮助您对 pod 安全解决方案做出更明智的决定。

Policy-as-code （与 Pod 安全标准相比）

优点：

- 更灵活、更精细（如有必要，可细化到资源属性）
- 不只是专注于 pod，还可用于不同的资源和操作
- 不只是应用于命名空间级别
- 比 Pod 安全标准更成熟
- 决策可以基于 API 服务器请求负载中的任何内容，以及现有的集群资源和外部数据（取决于解决方案）
- 支持在验证之前更改 API 服务器请求（取决于解决方案）
- 可以生成补充策略和 Kubernetes 资源（取决于解决方案——从 pod 策略中，Kyverno 可以为更高级别的控制器（例如部署）自动生成策略。 Kyverno 还可以使用生成规则生成额外的 Kubernetes 资源“当创建新资源或更新源代码时”。 ）
- 可用于在调用 Kubernetes API 服务器之前向左移动，进入 CICD 管道（取决于解决方案）
- 可用于实现不一定与安全相关的行为，例如最佳实践、组织标准等。
- 可以在非 Kubernetes 用例中使用（取决于解决方案）
- 由于灵活性，用户体验可以根据用户的需求进行调整

缺点：

- 未内置于 Kubernetes 中
- 学习、配置和支持更复杂
- 策略制定可能需要新 skills/languages的 /功能

Pod 安全准入（与策略即代码相比）

优点：

- 内置于 Kubernetes 中
- 配置更简单
- 没有新的语言可供使用，也没有政策可供创作
- 如果集群的默认准入级别配置为特权，则可以使用命名空间标签将命名空间选择到容器安全配置文件中。

缺点：

- 不如策略即代码那样灵活或精细
- 只有 3 个级别的限制
- 主要集中在吊舱上

Summary

如果你目前除了 PSP 之外还没有 pod 安全解决方案，并且你需要的 Pod 安全态势符合 Pod 安全标准 (PSS) 中定义的模型，那么更简单的方法可能是采用 PSS 来代替策略即代码解决方案。但是，如果您的 pod 安全态势不符合 PSS 模型，或者您设想在 PSS 定义的控件之外添加其他控件，那么策略即代码解决方案似乎更合适。

# 建议

## 使用多个 Pod 安全准入 (PSA) 模式以获得更好的用户体验

如前所述，PSA 强制模式可防止应用具有 PSS 违规的 pod，但不会阻止更高级别的控制器，例如部署。实际上，部署将在没有任何迹象表明无法应用 pod 的情况下成功应用。虽然你可以使用 kubectl 来检查 Deployment 对象，并从 PSA 中发现失败的 pod 消息，但用户体验可能会更好。为了改善用户体验，应使用多种PSA模式（审计、强制执行、警告）。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

在上面的示例中，在定义了强制模式的情况下，当尝试将相应 PodSpec 中存在违规的 PSS 的部署清单应用于 Kubernetes API 服务器时，部署将成功应用，但是 pod 不会成功应用。而且，由于还启用了审核和警告模式，API 服务器客户端将收到一条警告消息，API 服务器审核日志事件也将添加一条消息注释。

### 限制可以以特权身份运行的容器

如前所述，以特权身份运行的容器会继承主机上分配给 root 的所有 Linux 功能。容器很少需要这些类型的权限才能正常运行。有多种方法可用于限制容器的权限和功能。

> **Important**
>
> Fargate 是一种启动类型，可让您运行 “无服务器” 容器，其中 Pod 的容器在 AWS 管理的基础设施上运行。使用 Fargate，你无法运行特权容器，也无法将你的容器配置为使用 HostNetwork 或 HostPort。

### 不要以 root 用户身份在容器中运行进程

默认情况下，所有容器都以 root 身份运行。如果攻击者能够利用应用程序中的漏洞并获得 shell 访问正在运行的容器的权限，则可能会出现问题。您可以通过多种方式降低这种风险。首先，从容器镜像中移除 shell。其次，将 USER 指令添加到你的 Dockerfile 中，或者以非 root 用户身份运行容器中的容器。Kubernetes PodSpec 在下面包含一组字段`spec.securityContext`，允许您指定运行应用程序的用户 and/or 组。这些字段`runAsGroup`分别是`runAsUser`和。

为了在 Kubernetes PodSpec 中强制使用及其相关元素，可以在集群中添加策略即代码或 Pod 安全标准。`spec.securityContext`这些解决方案允许您编写and/or 使用策略或配置文件，这些策略或配置文件可以在入站 Kubernetes API 服务器请求有效负载保存到 etcd 中之前对其进行验证。此外，策略即代码解决方案可以改变入站请求，在某些情况下还会生成新的请求。

### 切勿在 Docker 中运行 Docker 或将套接字安装到容器中

虽然这可以方便地在 Docker 容器中查看 build/run 镜像，但你基本上是将节点的完全控制权交给容器中运行的进程。如果你需要在 Kubernetes 上构建容器镜像，可以改用 Kaniko、builda h 或类似的构建服务。CodeBuild

> **Note**
>
> 用于 CICD 处理（例如构建容器镜像）的 Kubernetes 集群应与运行更广泛的工作负载的集群隔离开来。

## 限制 HostPath 的使用，或者如果需要 HostPath，则限制可以使用哪些前缀并将该卷配置为只读

hostPath是一个将目录从主机直接挂载到容器的卷。Pod 很少需要这种访问权限，但如果需要，则需要注意风险。默认情况下，以 root 身份运行的 pod 将拥有对 HostPath 公开的文件系统的写入权限。这可能允许攻击者修改 kubelet 设置，创建指向 HostPath 未直接暴露的目录或文件的符号链接，例如/，安装 ssh 密钥etc/shadow，读取挂载到主机的机密以及其他恶意内容。要降低来自 HostPath 的风险，请将配置spec.containers.volumeMounts为readOnly，例如：

```
volumeMounts:
- name: hostPath-volume
    readOnly: true
    mountPath: /host-path
```

您还应该使用策略即代码解决方案来限制hostPath卷可以使用的目录，或者完全禁止hostPath使用。您可以使用 Pod 安全标准基准或受限策略来防止使用hostPath。

有关特权升级危险的更多信息，请阅读塞思·阿特的博客 Ba d Pods：Kubernetes Pod 权限升级。

## 为每个容器设置请求和限制，以避免资源争用和 DoS 攻击

从理论上讲，没有请求或限制的 Pod 会消耗主机上的所有可用资源。当其他 Pod 被调度到一个节点上时，该节点可能会遇到 CPU 或内存压力，这可能导致 Kubelet 终止或将 Pod 从该节点中驱逐出去。虽然您无法阻止这种情况同时发生，但设置请求和限制将有助于最大限度地减少资源争用，并降低因应用程序编写不当而消耗过多资源所带来的风险。

podSpec允许您指定 CPU 和内存的请求和限制。CPU 被视为可压缩资源，因为它可能被超额订阅。内存是不可压缩的，也就是说，它不能在多个容器之间共享。

当你指定 CPU 或内存请求时，你实际上是在指定容器保证获得的内存量。Kubernetes 会汇总一个 Pod 中所有容器的请求，以确定将 Pod 调度到哪个节点上。如果容器超过请求的内存量，则如果节点上存在内存压力，则该容器可能会被终止。

限制是容器允许消耗的最大 CPU 和内存资源量，直接对应于为容器创建的 cgroup 的`memory.limit_in_bytes`值。超过内存限制的容器将被OOM杀死。如果容器超出其 CPU 限制，它将受到限制。

> **Note**
>
> 使用容器时`resources.limits`，强烈建议容器资源使用量（又名资源足迹）以数据为导向，并根据负载测试进行准确。如果没有准确且值得信赖的资源占用空间，则`resources.limits`可以填充容器。例如，`resources.limits.memory`可以比可观察的最大值高 20-30%，以考虑潜在的内存资源限制不准确性。

Kubernetes 使用三个服务质量 (QoS) 类别来确定节点上运行的工作负载的优先级。这些方法包括：

- 保证
- 可爆发
- 尽力而为

如果未设置限制和请求，则将 Pod 配置为尽力而为（最低优先级）。 Best-effort 当内存不足时，pod 是第一个被杀死的。如果对 Pod 中的所有容器设置了限制，或者将请求和限制设置为相同的值而不等于 0，则该 Pod 将被配置为保证（最高优先级）。除非超过其配置的内存限制，否则保证的 pod 不会被杀死。如果限制和请求配置为不同的值且不等于 0，或者容器中的一个容器设置了限制，而其他容器没有或对不同的资源设置了限制，则这些 Pod 将被配置为可突发性（中等优先级）。这些 pod 有一些资源保证，但是一旦它们超过了请求的内存，它们就会被杀死。

> **Important**
>
> 请求不会影响容器的 cgroup 的`memory_limit_in_bytes`值；cgroup 限制设置为主机上的可用内存量。尽管如此，如果节点承受内存压力，将请求值设置得太低可能会导致 kubelet 将 pod 作为终止的目标。

| 类 | 优先级 | 条件 | 击杀条件 |
| --- | --- | --- | --- |
| 能保证 | 最高 | 限制 = 请求！ = 0 | 仅超过内存限制 |

| 类 | 优先级 | 条件 | 击杀条件 |
| --- | --- | --- | --- |
| 可突增 | medium | 极限！ = 请求！ = 0 | 如果超出请求内存，则可能被杀死 |
| Best-Effort | 最低 | 限制和请求未设置 | 第一个在内存不足时被杀死 |

有关资源 QoS 的更多信息，请参阅 Kub ernetes 文档。

您可以通过在命名空间上设置资源配额或创建限制范围来强制使用请求和限制。资源配额允许您指定分配给命名空间的资源总量，例如 CPU 和 RAM。将其应用于命名空间时，它会强制您为部署到该命名空间中的所有容器指定请求和限制。相比之下，限制范围使您可以更精细地控制资源分配。通过限制范围，您可以 min/max 限制命名空间内每个 pod 或每个容器的 CPU 和内存资源。如果未提供默认 request/limit 值，也可以使用它们来设置默认值。

Policy-as-code 解决方案可以用来强制执行请求和限制。甚至可以在创建命名空间时创建资源配额和限制范围。

### 不允许权限升级

特权升级允许进程更改其运行所处的安全上下文。Sudo 就是一个很好的例子，带有 SUID 或 SGID 位的二进制文件也是如此。权限升级基本上是用户以其他用户或组的权限执行文件的一种方式。您可以通过实施设置为`false`或在中设置`securityContext.allowPrivilegeEscalation`的策略即代码变更策略`allowPrivilegeEscalation`来防止容器使用特权升级。podSpec Policy-as-code 如果检测到设置不正确，也可以使用策略来防止 API 服务器请求成功。Pod 安全标准还可用于防止 Pod 使用权限升级。

### 禁用 ServiceAccount 令牌挂载

对于不需要访问 Kubernetes API 的 Pod，你可以禁用在容器规范上自动挂载 ServiceAccount 令牌，也可以禁用所有使用特定的 Pod 的自动挂载令牌。 ServiceAccount

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-automount
```

```
spec:
  automountServiceAccountToken: false
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-no-automount
automountServiceAccountToken: false
```

## 禁用服务发现

对于不需要查找或调用集群内服务的 Pod，您可以减少提供给 Pod 的信息量。你可以将 Pod 的 DNS 策略设置为不使用 CoreDNS，也不要将 Pod 命名空间中的服务作为环境变量公开。有关服务链接的更多信息，请参阅有关环境变量的 Kubernetes 文档。Pod 的 DNS 策略的默认值是 “ClusterFirst”，它使用集群内 DNS，而非默认值 “默认” 使用底层节点的 DNS 解析。有关更多信息，请参阅有关 Pod DNS 政策的 Kubernetes 文档。

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-service-info
spec:
    dnsPolicy: Default # "Default" is not the true default value
    enableServiceLinks: false
```

## 使用只读的根文件系统配置您的图像

使用只读根文件系统配置图像可防止攻击者覆盖应用程序使用的文件系统上的二进制文件。如果您的应用程序必须写入文件系统，请考虑写入临时目录或连接并装入一个卷。你可以通过按 SecurityContext 如下方式设置 pod 来强制执行此操作：

```
...
securityContext:
  readOnlyRootFilesystem: true
...
```

Policy-as-code 并且 Pod 安全标准可用于强制执行此行为。

Example

根据 Windows规定，true对于在Windows上运行的容器，Kubernetes 中的容器securityContext.readOnlyRootFilesystem不能设置为，因为注册表和系统进程需要写入权限才能在容器内运行。

## 工具和资源

- Amazon EKS 安全沉浸式研讨会——Pod 安全
- op@@ enpolic agent/gatekeeper y-library：OPA Gatekeeper 策略库是一个 OPA/Gatekeeper 策略库，你可以用它来替代 PSP。
- Kyverno 政策库
- EKS 的常用 OPA 和 Kyverno 政策的集合。
- 基于政策的对策：第 1 部分
- 基于政策的对策：第 2 部分
- Pod Security Policy Migrator 是一种将 PSP 转换为 OPA/Gatekeeper KubeWarden、或 Kyverno 策略的工具
- NeuVector 由 SUSE 开源提供的零信任容器安全平台提供进程和文件系统策略以及准入控制规则。

# 租户隔离

当我们想到多租户时，我们通常希望将用户或应用程序与在共享基础架构上运行的其他用户或应用程序隔离开来。

Kubernetes 是一个单租户协调器，即控制平面的单个实例由集群中的所有租户共享。但是，你可以使用各种各样的 Kubernetes 对象来创建多租户的外观。例如，可以实施命名空间和Role-based 访问控制 (RBAC)，以便在逻辑上隔离租户彼此。同样，配额和限制范围可用于控制每个租户可以消耗的集群资源量。尽管如此，集群是唯一能提供强大安全边界的结构。这是因为设法获得对集群内主机的访问权限的攻击者可以检索安装在该主机上的所有 Secrets 和 Volumes。ConfigMaps他们还可以模仿 Kubelet，这样他们就可以操纵在集群内横向 and/or 移动的节点的属性。

以下各节将解释如何实现租户隔离，同时降低使用像 Kubernetes 这样的单一租户协调器的风险。

# 软多租户

在软多租户中，您可以使用原生 Kubernetes 结构，例如命名空间、角色和角色绑定以及网络策略，在租户之间创建逻辑分离。例如，RBAC 可以阻止租户访问或操纵彼此的资源。配额和限制范围控制每个租户可以消耗的集群资源量，而网络策略可以帮助防止部署到不同命名空间的应用程序相互通信。

但是，这些控件都无法阻止来自不同租户的 Pod 共享节点。如果需要更强的隔离，你可以使用节点选择器、反亲和性规则、 and/or 污点和容忍度来强制将来自不同租户的 Pod 调度到单独的节点（通常称为单租户节点）。在租户众多的环境中，这可能会变得相当复杂，而且成本高得令人望而却步。

> **Important**
>
> 使用命名空间实现的软多租户不允许您向租户提供经过筛选的命名空间列表，因为命名空间是一种全局范围的类型。如果租户能够查看特定的命名空间，则可以查看集群中的所有命名空间。

> **Warning**
>
> 借助软多租户，默认情况下，租户可以查询 CoreDNS 以获取集群内运行的所有服务。攻击者可以通过 `..svc.cluster.local`从集群中的任何 pod 运行 dig SRV 来利用此漏洞。如果您需要限制访问集群中运行的服务的 DNS 记录，请考虑使用适用于 CoreDNS 的防火墙或策略插件。有关更多信息，请参阅 https://github.com/coredns/policy #kubernetes-metadata-multi-tenancy-policy。

Kiosk 是一个开源项目，可以帮助实现软多租户。它以一系列提供以下功能的 CRD 和控制器形式实现：

- 用于在共享 Kubernetes 集群中分离租户的@@ 账户和账户用户
- Self-Service 为账户用户@@ 配置命名空间
- 账户限制可确保共享集群时的服务质量和公平性
- 用于安全租户隔离和自助服务@@ 命名空间初始化的命名空间模板

Loft 是 Kiosk 维护者提供的商业产品 DevSpace，它增加了以下功能：

- Multi-cluster 访问权限，用于授予对不同集群中空间的访问权限

- 睡眠模式可在空间处于非活动状态时缩小部署规模
- 使用 OIDC 身份@@ 验证提供商进行单点登录，例如 GitHub

软多租户可以解决三个主要用例。

### 企业设置

第一种是在企业环境中，“租户” 是半信任的，因为他们是员工、承包商或以其他方式获得组织授权。每个租户通常会与行政部门（例如部门或团队）保持一致。

在这种类型的设置中，集群管理员通常负责创建命名空间和管理策略。他们还可以实现委托管理模式，让某些人监督命名空间，允许他们对非策略相关对象（如部署、服务、pod、作业等）执行 CRUD 操作。

在此设置中，容器运行时提供的隔离可能是可以接受的，或者可能需要通过其他控制来增强 pod 安全性。如果需要更严格的隔离，可能还需要限制不同命名空间中的服务之间的通信。

### Kubernetes 即服务

相比之下，软多租户可以在你想要提供 Kubernetes 即服务 (KaaS) 的环境中使用。使用 KaaS，您的应用程序与提供一组 PaaS 服务的控制器和 CRD 一起托管在共享集群中。租户直接与 Kubernetes API 服务器交互，并允许租户对非策略对象执行 CRUD 操作。还有一个自助服务要素，即可以允许租户创建和管理自己的命名空间。在这种类型的环境中，假定租户正在运行不受信任的代码。

要隔离此类环境中的租户，您可能需要实施严格的网络策略和容器沙箱。沙盒是你在 Firecracker 等微型虚拟机或用户空间内核中运行 pod 容器的地方。现在，你可以使用 EKS Fargate 创建沙盒吊舱。

### 软件即服务 (SaaS)

软多租户的最终用例是（ Software-as-a-ServiceSaaS）设置。在此环境中，每个租户都与集群中运行的应用程序的特定实例相关联。每个实例通常都有自己的数据，并使用单独的访问控制，这些控制通常独立于 Kubernetes RBAC。

与其他用例不同，SaaS 设置中的租户不直接与 Kubernetes API 交互。相反，SaaS 应用程序负责与 Kubernetes API 接口，以创建支持每个租户所需的对象。

## Kubernetes 构造

在每种情况下，都使用以下结构将租户彼此隔离开来：

## 命名空间

命名空间是实现软多租户的基础。它们允许您将集群划分为逻辑分区。实现多租户所需的配额、网络策略、服务帐号和其他对象限定在命名空间内。

## 网络策略

默认情况下，允许 Kubernetes 集群中的所有 Pod 相互通信。使用网络策略可以改变这种行为。

网络策略使用标签或 IP 地址范围限制 Pod 之间的通信。在需要在租户之间进行严格网络隔离的多租户环境中，我们建议从一个默认规则开始，即拒绝 Pod 之间的通信，另一个允许所有 Pod 查询 DNS 服务器以获取名称解析的规则。有了这些，你就可以开始添加更宽松的规则了，允许在命名空间内进行通信。这可以根据需要进一步完善。

> **Note**
>
> 在 AWS 上运行 Kubernetes 时，Amazon VPC CNI 现在支持 Kubernetes 网络策略，以创建可以隔离敏感工作负载并保护它们免受未经授权的访问的策略。这意味着您可以在 Amazon EKS 集群中使用网络策略 API 的所有功能。这种精细控制级别使您能够实现最小权限原则，从而确保只有经过授权的 Pod 才能相互通信。

> **Important**
>
> 网络策略是必要的，但还不够。网络策略的实施需要诸如 Calico 或 Cilium 之类的策略引擎。

## Role-based 访问控制 (RBAC)

角色和角色绑定是 Kubernetes 对象，用于在 Kubernetes 中强制执行基于角色的访问控制 (RBAC)。角色包含可以对集群中的对象执行的操作列表。角色绑定指定角色适用的个人或群组。在企业和 KaaS 设置中，RBAC 可用于允许选定的群组或个人管理对象。

## 配额

配额用于定义集群中托管的工作负载的限制。使用配额，您可以指定 Pod 可以消耗的最大 CPU 和内存量，也可以限制可以在集群或命名空间中分配的资源数量。限制范围允许您声明每个限制的最小值、最大值和默认值。

在共享集群中过度使用资源通常是有益的，因为它可以最大限度地利用资源。但是，不受限制地访问集群会导致资源短缺，从而导致性能下降和应用程序可用性丧失。如果 Pod 的请求设置得太低并且实际资源利用率超过节点的容量，则该节点将开始承受 CPU 或内存压力。发生这种情况时，Pod 可能会被重新启动 and/or 并逐出节点。

为了防止这种情况发生，您应该计划对多租户环境中的命名空间施加配额，以强制租户在集群上调度 pod 时指定请求和限制。它还可以通过限制 Pod 可以消耗的资源量来缓解潜在的拒绝服务。

您还可以使用配额来分配集群的资源，使其与租户的支出保持一致。这在 KaaS 场景中特别有用。

### Pod 优先级和抢占权

当你想让 Pod 相对于其他 Pod 更重要时，Pod 优先级和抢占会很有用。例如，使用容器优先级，您可以将客户 A 的 Pod 配置为比客户 B 更高的优先级运行。当可用容量不足时，调度器会将优先级较低的 pod 从客户 B 中驱逐出来，以容纳客户 A 的高优先级 Pod。这在 SaaS 环境中特别方便，在这种环境中，愿意支付溢价的客户会获得更高的优先级。

> **Important**
>
> Pod 优先级可能会对其他优先级较低的 Pod 产生不良影响。例如，尽管受害的 pod 会优雅地终止，但不能保证，这可能会使依赖法定数量的 Pod 优先级较低的应用程序崩溃，但请参见抢占限制。 PodDisruptionBudget

## 缓解控制措施

作为多租户环境的管理员，你最关心的是防止攻击者访问底层主机。应考虑采取以下控制措施来降低这种风险：

### 容器的沙盒化执行环境

沙箱是一种技术，通过这种技术，每个容器都在自己的独立虚拟机中运行。执行吊舱沙箱的技术包括 Firecracker 和 Weave 的 Fire kube。

有关让 Firecracker 成为 EKS 支持的运行时所做的努力的更多信息，请参阅 https://threadreaderapp.com/thread/1238496944684597248.html.

## 开放策略代理 (OPA) & 看门人

Gatekeeper 是一个 Kubernetes 准入控制器，用于强制执行使用 OPA 创建的策略。借助 OPA，您可以创建一个策略，在单独的实例上运行租户的 Pod，或者以比其他租户更高的优先级运行 Pod。可以在该项目的 GitHub存储库中找到常用 OPA 策略的集合。

还有一个适用于 CoreDNS 的实验性 OPA 插件，允许你使用 OPA 来处理 CoreDNS 返回 filter/control 的记录。

## Kyverno

Kyverno 是一个 Kubernetes 原生策略引擎，它可以使用策略作为 Kubernetes 资源来验证、变异和生成配置。Kyverno 使用 Kustomize-style 叠加层进行验证，支持 JSON 补丁和用于突变的战略合并补丁，并且可以基于灵活的触发器跨命名空间克隆资源。

您可以使用 Kyverno 来隔离命名空间、强制执行 pod 安全和其他最佳实践，并生成网络策略等默认配置。该项目的 GitHub存储库中包含几个示例。Kyverno 网站上的政策库中包含了许多其他内容。

## 将租户工作负载隔离到特定节点

将租户工作负载限制在特定节点上运行可用于增强软多租户模型中的隔离。使用这种方法，租户特定的工作负载只能在为相应租户配置的节点上运行。为了实现这种隔离，使用原生 Kubernetes 属性（节点亲和性以及污点和容忍）来定位特定节点进行 Pod 调度，并防止来自其他租户的 Pod 被调度到租户特定的节点上。

第 1 部分-节点亲和力

Kubernetes 节点亲和力用于根据节点标签定位节点进行调度。使用节点亲和性规则，Pod 会被吸引到与选择器术语匹配的特定节点。在以下 pod 规范中，`requiredDuringSchedulingIgnoredDuringExecution`节点亲和性应用于相应的 pod。结果是 pod 将瞄准标有以下内容的节点 key/value:`node-restriction.kubernetes.io/tenant: tenants-x`.

```
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
```

```
          - key: node-restriction.kubernetes.io/tenant
            operator: In
            values:
            - tenants-x
...
```

有了这种节点亲和性，在调度期间需要标签，但在执行期间不需要；如果底层节点的标签发生变化，则不会仅仅因为标签更改而驱逐这些 Pod。但是，未来的日程安排可能会受到影响。

> **Warning**
>
> 的标签前缀在 Kubernetes 中`node-restriction.kubernetes.io/`具有特殊含义。NodeRestriction为 EKS 集群启用，可`kubelet`防止 adding/removing /更新带有此前缀的标签。攻击者无法使用 a `kubelet's credentials to update the node object or modify the system setup to pass these labels into `kubelet s kubelet` 不允许修改这些标签。如果将此前缀用于所有 Pod 到节点的调度，则可以防止攻击者可能想要通过修改节点标签将一组不同的工作负载吸引到节点的情况。

Example

我们可以使用节点选择器来代替节点亲和性。但是，节点亲和性更具表现力，允许在 Pod 调度期间考虑更多条件。有关差异和更高级的调度选择的更多信息，请参阅这篇关于高级 Kubernetes pod 到节点调度的CNCF博客文章。

第 2 部分-污点和容忍

将 pod 吸引到节点只是这种由三部分组成的方法的第一部分。为了使这种方法起作用，我们必须阻止 Pod 调度到未获授权的节点上。为了击退不需要的或未经授权的 pod，Kubernetes 使用节点污点。污点用于在节点上设置阻止 Pod 被调度的条件。下面的污点使用键值对。`tenant: tenants-x`

```
...
    taints:
      - key: tenant
        value: tenants-x
        effect: NoSchedule
...
```

鉴于上述节点`taint`，只有容忍污点的 Pod 才允许在该节点上调度。为了允许将已授权的 pod 调度到节点上，相应的 pod 规格必须包含`toleration`对污点的 a，如下所示。

```
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

具有上述内容的 Pod `toleration` 不会停止在节点上调度，至少不会因为这种特定的污点而停止。在某些情况下（例如节点资源压力），Kubernetes 还使用污点来暂时停止 Pod 的调度。通过节点亲和性以及污点和容忍度，我们可以有效地将所需的 Pod 吸引到特定节点并击退不需要的 Pod。

> **Important**
>
> 某些 Kubernetes 容器需要在所有节点上运行。这些 pod 的例子是容器网络接口 (CNI) 和 kube-proxy 守护程序集启动的 pod。为此，这些 pod 的规格包含非常宽松的容忍度，可以容忍不同的污点。应注意不要改变这些容忍度。更改这些容忍度可能会导致集群操作不正确。此外，策略管理工具（例如OPA/Gatekeeper和 Kyverno）可用于编写验证策略，防止未经授权的 pod 使用这些允许容忍度。

第 3 部分-节点选择的 Policy-based 管理

有多种工具可用于帮助管理 Pod 规范的节点亲和性和容忍度，包括在 CICD 管道中强制执行规则。但是，还应在 Kubernetes 集群级别强制执行隔离。为此，可以使用策略管理工具根据请求有效载荷对入站 Kubernetes API 服务器请求进行变更，以应用上述相应的节点关联性规则和容忍度。

例如，可以为发往 tenants-x 命名空间的 Pod 加上正确的节点亲和力和容忍度，以允许在 tenants-x 节点上进行调度。利用使用 Kubernetes 变更准入 Webhook 配置的策略管理工具，可以使用策略来改变入站 Pod 的规范。这些突变添加了所需的元素以允许所需的调度。添加节点亲和性的示例 OPA/Gatekeeper 策略如下所示。

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-nodeaffinity-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds Node affinity - https://kubernetes.io/docs/concepts/scheduling-eviction/
assign-pod-node/#node-affinity
```

```
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location:
 "spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms"
  parameters:
    assign:
      value:
        - matchExpressions:
          - key: "tenant"
            operator: In
            values:
            - "tenants-x"
```

上述策略应用于 Kubernetes API 服务器请求，即将 pod 应用于 tenants-x 命名空间。该策略添加了`requiredDuringSchedulingIgnoredDuringExecution`节点亲和性规则，这样 Pod 就会被带`tenant: tenants-x`标签的节点所吸引。

第二个策略（如下所示）使用与目标命名空间和组、种类和版本相同的匹配标准，将容忍度添加到相同的 pod 规格中。

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-toleration-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds toleration - https://kubernetes.io/docs/concepts/scheduling-eviction/taint-
and-toleration/
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location: "spec.tolerations"
  parameters:
    assign:
```

```
      value:
      - key: "tenant"
        operator: "Equal"
        value: "tenants-x"
        effect: "NoSchedule"
```

上述策略特定于 pod；这是由于策略`location`元素中变异元素的路径造成的。可以编写其他策略来处理创建 pod 的资源，例如 Deployment 和 Job 资源。列出的政策和其他示例可以在本指南的配套GitHub项目中看到。

这两个突变的结果是，Pod 被所需的节点所吸引，同时不会被特定的节点污点所排斥。为了验证这一点，我们可以看到两次`kubectl`调用的输出片段，以获取标有标签的节点`tenant=tenants-x`，然后在`tenants-x`命名空间中获取 pod。

```
kubectl get nodes -l tenant=tenants-x
NAME
ip-10-0-11-255...
ip-10-0-28-81...
ip-10-0-43-107...

kubectl -n tenants-x get pods -owide
NAME                                  READY   STATUS    RESTARTS   AGE   IP
 NODE
tenant-test-deploy-58b895ff87-2q7xw   1/1     Running   0          13s   10.0.42.143
 ip-10-0-43-107...
tenant-test-deploy-58b895ff87-9b6hg   1/1     Running   0          13s   10.0.18.145
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-nxvw5   1/1     Running   0          13s   10.0.30.117
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-vw796   1/1     Running   0          13s   10.0.3.113
 ip-10-0-11-255...
tenant-test-pod                       1/1     Running   0          13s   10.0.35.83
 ip-10-0-43-107...
```

从上面的输出中我们可以看出，所有的 pod 都被调度到标有标注的节点上`tenant=tenants-x`。简而言之，Pod 只能在所需的节点上运行，而其他 pod（没有所需的亲和力和容忍度）则不会。租户工作负载被有效隔离。

变异的 pod 规范示例如下所示。

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: tenant-test-pod
  namespace: tenants-x
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: tenant
            operator: In
            values:
            - tenants-x
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

**Important**

Policy-management 集成到 Kubernetes API 服务器请求流中的工具，使用变更和验证准入 webhook，旨在在指定的时间范围内响应 API 服务器的请求。这通常是 3 秒或更短。如果 webhook 调用未能在配置的时间内返回响应，则入站 API 服务器请求的变异 and/or 验证可能会发生，也可能不会发生。此行为取决于准入 webhook 配置设置为 “失效打开” 还是 “失效关闭”。

在上面的示例中，我们使用了为编写的策略 OPA/Gatekeeper。但是，还有其他策略管理工具可以处理我们的节点选择用例。例如，此 Kyverno 策略可用于处理节点亲和性突变。

**Note**

如果操作正常，变更策略将影响入站 API 服务器请求有效负载的所需更改。但是，还应包括验证策略，以验证在允许更改持续存在之前，是否发生了所需的更改。当使用这些策略进行租户到节点隔离时，这一点尤其重要。另外一个好主意是加入审计策略，以便定期检查集群中是否有不需要的配置。

### 参考

- k-rail 旨在通过执行某些策略来帮助您保护多租户环境。
- 使用 Amazon EK MultiTenant S 的 SaaS 应用程序的安全实践

## 艰难的多租户

硬多租户可以通过为每个租户配置单独的集群来实现。虽然这在租户之间提供了非常牢固的隔离，但它也有一些缺点。

首先，当你有很多租户时，这种方法很快就会变得昂贵。您不仅需要为每个集群支付控制平面成本，而且无法在集群之间共享计算资源。这最终会导致碎片化，其中一部分集群未得到充分利用，而其他集群则被过度利用。

其次，您可能需要购买或构建特殊工具来管理所有这些集群。随着时间的推移，管理成百上千个集群可能会变得过于繁琐。

最后，与创建命名空间相比，为每个租户创建集群会很慢。但是，在高度监管的行业或需要严格隔离的SaaS环境中，可能需要采取硬租赁方法。

## 未来的方向

Kubernetes 社区已经认识到软多租户当前的缺点以及硬多租户面临的挑战。Multi-Tenancy 特别兴趣小组 (SIG) 正试图通过多个孵化项目来解决这些缺点，包括分层命名空间控制器 (HNC) 和虚拟集群。

HNC 提案 (KEP) 描述了一种在命名空间之间创建父子关系的方法，该方法具有 [policy] 对象继承，并允许租户管理员创建子命名空间。

虚拟集群提案描述了一种机制，用于为集群中的每个租户（也称为 “Kubernetes 上的 Kubernetes on Kubernetes”）创建单独的控制平面服务实例，包括 API 服务器、控制器管理器和调度器。

Multi-Tenancy基准测试提案提供了使用命名空间进行隔离和分段共享集群的指南，以及用于验证是否符合指南的命令行工具 kubectl-mtb。

## Multi-cluster 管理工具和资源

- 万岁云
- Kommander
- 镜头
- Nirmata

- Rafay
- 牧场主
- Weave Flux

# 审计和日志

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

出于各种不同的原因，收集和分析 [审计] 日志非常有用。日志可以帮助进行根本原因分析和归因，即将更改归因于特定用户。收集到足够的日志后，它们也可以用来检测异常行为。在 EKS 上，审核日志将发送到 Amazon Cloudwatch 日志。EKS 的审计政策如下：

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
  # Log full request and response for changes to aws-auth ConfigMap in kube-system
 namespace
  - level: RequestResponse
    namespaces: ["kube-system"]
    verbs: ["update", "patch", "delete"]
    resources:
      - group: "" # core
        resources: ["configmaps"]
        resourceNames: ["aws-auth"]
    omitStages:
      - "RequestReceived"
  # Do not log watch operations performed by kube-proxy on endpoints and services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
      - group: "" # core
        resources: ["endpoints", "services", "services/status"]
  # Do not log get operations performed by kubelet on nodes and their statuses
  - level: None
    users: ["kubelet"] # legacy kubelet identity
    verbs: ["get"]
```

```
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get operations performed by the system:nodes group on nodes and their
 statuses
  - level: None
    userGroups: ["system:nodes"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get and update operations performed by controller manager, scheduler,
 and endpoint-controller on endpoints in kube-system namespace
  - level: None
    users:
      - system:kube-controller-manager
      - system:kube-scheduler
      - system:serviceaccount:kube-system:endpoint-controller
    verbs: ["get", "update"]
    namespaces: ["kube-system"]
    resources:
      - group: "" # core
        resources: ["endpoints"]
  # Do not log get operations performed by apiserver on namespaces and their statuses/
finalizations
  - level: None
    users: ["system:apiserver"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["namespaces", "namespaces/status", "namespaces/finalize"]
  # Do not log get and list operations performed by controller manager on
 metrics.k8s.io resources
  - level: None
    users:
      - system:kube-controller-manager
    verbs: ["get", "list"]
    resources:
      - group: "metrics.k8s.io"
  # Do not log access to health, version, and swagger non-resource URLs
  - level: None
    nonResourceURLs:
      - /healthz*
      - /version
```

```
      - /swagger*
  # Do not log events resources
  - level: None
    resources:
      - group: "" # core
        resources: ["events"]
  # Log request for updates/patches to nodes and pods statuses by kubelet and node
 problem detector
  - level: Request
    users: ["kubelet", "system:node-problem-detector", "system:serviceaccount:kube-
system:node-problem-detector"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log request for updates/patches to nodes and pods statuses by system:nodes group
  - level: Request
    userGroups: ["system:nodes"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log delete collection requests by namespace-controller in kube-system namespace
  - level: Request
    users: ["system:serviceaccount:kube-system:namespace-controller"]
    verbs: ["deletecollection"]
    omitStages:
      - "RequestReceived"
  # Log metadata for secrets, configmaps, and tokenreviews to protect sensitive data
  - level: Metadata
    resources:
      - group: "" # core
        resources: ["secrets", "configmaps"]
      - group: authentication.k8s.io
        resources: ["tokenreviews"]
    omitStages:
      - "RequestReceived"
  # Log requests for serviceaccounts/token resources
  - level: Request
    resources:
```

```
    - group: "" # core
      resources: ["serviceaccounts/token"]
# Log get, list, and watch requests for various resource groups
- level: Request
  verbs: ["get", "list", "watch"]
  resources:
    - group: "" # core
    - group: "admissionregistration.k8s.io"
    - group: "apiextensions.k8s.io"
    - group: "apiregistration.k8s.io"
    - group: "apps"
    - group: "authentication.k8s.io"
    - group: "authorization.k8s.io"
    - group: "autoscaling"
    - group: "batch"
    - group: "certificates.k8s.io"
    - group: "extensions"
    - group: "metrics.k8s.io"
    - group: "networking.k8s.io"
    - group: "policy"
    - group: "rbac.authorization.k8s.io"
    - group: "scheduling.k8s.io"
    - group: "settings.k8s.io"
    - group: "storage.k8s.io"
  omitStages:
    - "RequestReceived"
# Default logging level for known APIs to log request and response
- level: RequestResponse
  resources:
    - group: "" # core
    - group: "admissionregistration.k8s.io"
    - group: "apiextensions.k8s.io"
    - group: "apiregistration.k8s.io"
    - group: "apps"
    - group: "authentication.k8s.io"
    - group: "authorization.k8s.io"
    - group: "autoscaling"
    - group: "batch"
    - group: "certificates.k8s.io"
    - group: "extensions"
    - group: "metrics.k8s.io"
    - group: "networking.k8s.io"
    - group: "policy"
    - group: "rbac.authorization.k8s.io"
```

```
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for all other requests to log metadata only
  - level: Metadata
    omitStages:
      - "RequestReceived"
```

## 建议

### 启用审核日志

审计日志是由 EKS 管理的 EKS 托管的 Kubernetes 控制平面日志的一部分。控制平面日志（包括 Kubernetes API 服务器、控制器管理器和调度器的日志以及审计日志）的说明可以在这里找到 #enabling-control-plane-log-export。 enabling/disabling https://docs.aws.amazon.com/eks/latest/userguide/control-plane-logs.html

> **Note**
>
> 启用控制平面日志记录后，将产生存储日志的 CloudWatch费用。这就引发了一个关于持续安全成本的更广泛问题。最终，您将不得不权衡这些成本与安全漏洞的成本，例如财务损失、声誉损害等。您可能会发现，只要实施本指南中的一些建议，就可以充分保护您的环境。

> **Warning**
>
> CloudWatch 日志条目的最大大小为 1MB，而 Kubernetes API 请求的最大大小为 1.5MiB。大于 1MB 的日志条目要么被截断，要么仅包含请求元数据。

### 利用审计元数据

Kubernetes 审计日志包含两个注释，用于指示请求是否获得授权`authorization.k8s.io/decision`以及做出决定的原因。`authorization.k8s.io/reason`使用这些属性来确定允许特定 API 调用的原因。

### 为可疑事件创建警报

创建警报，自动提醒您 403 个禁止响应和 401 个未授权响应增加的地方，然后使用诸如`hostsourceIPs`、和之类的属性`k8s_user.username`来找出这些请求的来源。

### 使用 “日志见解” 分析日志

使用 CloudWatch Log Insights 监控对 RBAC 对象的更改，例如角色、 RoleBindings ClusterRoles、和。 ClusterRoleBindings下面是一些示例查询：

列出以下内容的更新 `aws-auth` ConfigMap：

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["update", "patch"]
| filter objectRef.resource = "configmaps" and objectRef.name = "aws-auth" and
 objectRef.namespace = "kube-system"
| sort @timestamp desc
```

列出新建的验证 webhook 或对验证 webhook 的更改：

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["create", "update", "patch"] and responseStatus.code = 201
| filter objectRef.resource = "validatingwebhookconfigurations"
| sort @timestamp desc
```

列出角色的创建、更新、删除操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="roles" and verb in ["create", "update", "patch", "delete"]
```

列出创建、更新、删除操作 RoleBindings：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="rolebindings" and verb in ["create", "update", "patch",
 "delete"]
```

列出创建、更新、删除操作 ClusterRoles：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterroles" and verb in ["create", "update", "patch",
 "delete"]
```

列出创建、更新、删除操作 ClusterRoleBindings：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterrolebindings" and verb in ["create", "update",
 "patch", "delete"]
```

绘制针对密钥的未经授权的读取操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="secrets" and verb in ["get", "watch", "list"] and
 responseStatus.code="401"
| stats count() by bin(1m)
```

失败的匿名请求列表：

```
fields @timestamp, @message, sourceIPs.0
| sort @timestamp desc
| limit 100
| filter user.username="system:anonymous" and responseStatus.code in ["401", "403"]
```

## 审核您的 CloudTrail 日志

使用服务账户的 IAM 角色 (IRSA) 的 pod 调用的 AWS API 会自动与服务账户的名称 CloudTrail 一起登录到。如果未明确授权调用 API 的服务账号的名称出现在日志中，则可能表明 IAM 角色的信任策略配置错误。一般而言，Cloudtrail 是将 AWS API 调用归因于特定 IAM 委托人的好方法。

### 使用 CloudTrail Insights 发现可疑活动

CloudTrail insights 会自动分析来自CloudTrail 跟踪的写入管理事件，并提醒您注意异常活动。这可以帮助您确定您的 AWS 账户中写入 API 的调用量何时增加，包括来自使用 IRSA 担任 IAM 角色的 pod 的调用量。有关更多信息，请参阅发布CloudTrail 见解：识别和响应异常 API 活动。

### 其他资源

随着日志量的增加，使用 Log Insights 或其他日志分析工具解析和筛选日志可能会变得无效。作为替代方案，你可能需要考虑运行 Sysdi g Falco 和 ekscloudwat ch。Falco 会分析审计日志，并标记长时间内的异常或滥用行为。ekscloudwatch 项目将审计日志事件从 Falco 转发给 Falco CloudWatch 进行分析。Falco提供了一组默认的审计规则，并允许您添加自己的审计规则。

还有一种选择可能是将审核日志存储在 S3 中，然后使用 R SageMaker andom Cut Forest 算法来处理需要进一步调查的异常行为。

## 工具和资源

以下商业和开源项目可用于评估您的集群与既定最佳实践的一致性：

- Amazon EKS 安全沉浸式研讨会——侦探控件
- kubaudit
- kube-scan 根据 Kubernetes 通用配置评分系统框架为集群中运行的工作负载分配风险评分
- kubesec.io
- 北极星
- 右翼
- Snyk
- Kubescape Kubescape 是一款开源 kubernetes 安全工具，可以扫描集群、YAML 文件和 Helm 图表。它根据多个框架（包括NSA-CISA和 M ITRE ATT&C K®）检测错误配置

# 网络安全

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

网络安全有几个方面。第一种是应用限制服务之间网络流量的规则。第二种方法涉及对传输中的流量进行加密。在 EKS 上实施这些安全措施的机制多种多样，但通常包括以下几项：

## 交通管制

- 网络策略
- 安全组

## 网络加密

- 服务网格
- 入口控制器和负载均衡器
- Nitro 实例
- 带证书管理器的 ACM 私有 CA

## 网络政策

在 Kubernetes 集群中，默认情况下允许所有 Pod 到 Pod 的通信。尽管这种灵活性可能有助于促进实验，但它并不安全。Kubernetes 网络策略为您提供了一种机制，用于限制 Pod 之间（通常称为 East/West流量）以及 Pod 与外部服务之间的网络流量。Kubernetes 网络策略在 OSI 模型的第 3 层和第 4 层运行。网络策略使用 pod、命名空间选择器和标签来识别源和目标 Pod，但也可以包括 IP 地址、端口号、协议或这些内容的组合。网络策略可以应用于 pod 的入站或出站连接，通常称为入口和出口规则。

借助 Amazon VPC CNI 插件的原生网络策略支持，您可以实施网络策略来保护 kubernetes 集群中的网络流量。它与上游的 Kubernetes 网络策略 API 集成，可确保兼容和遵守 Kubernetes 标准。您可以使用上游 API 支持的不同标识符来定义策略。默认情况下，所有入口和出口流量都允许进入一个 Pod。当指定带有 PolicyType Ingress 的网络策略时，只有来自容器节点的连接和入口规则允许的连接才能连接到 Pod。同样适用于出口规则。如果定义了多个规则，则在做出决定时会考虑所有规则的并集。因此，评估顺序不会影响策略结果。

> **Important**
>
> 首次配置 EKS 集群时，默认情况下不启用 VPC CNI 网络策略功能。确保您部署了支持的 VPC CNI Add-on 版本，并在 vpc-cni 插件`true`上将`ENABLE_NETWORK_POLICY`标志设置为以启用此功能。有关详细说明，请参阅 Amazon EKS 用户指南。

# 建议

## 网络策略入门-遵循最低权限原则

### 创建默认拒绝策略

与 RBAC 策略一样，建议在网络策略中遵循最低权限访问原则。首先创建一个 “全部拒绝” 策略，该策略限制命名空间中的所有入站和出站流量。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

默认拒绝

| | | Target | | Source | | | | |
|---|---|---|---|---|---|---|---|---|
| # | Policy name | Namespace | Pods | Namespace | Pods | Subnet | Ports | |
| 1 | default-deny | default | Any | Any | Any | Any | Any | |

> **Note**
>
> 上图由 Tufin 的网络策略查看器创建。

### 创建允许 DNS 查询的规则

一旦设置了默认的 “全部拒绝” 规则，就可以开始对其他规则进行分层，例如允许 pod 查询 CoreDNS 以获取名称解析的规则。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-access
```

```
  namespace: default
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
    - protocol: UDP
      port: 53
```

允许 dns 访问

| | | Target | | Destination | | | | |
|---|---|---|---|---|---|---|---|---|
| # | Policy name | Namespace | Pods | Namespace | Pods | Subnet | Ports | |
| 1 | allow-dns-access | default | Any | name: kube-system | Any | | UDP: 53 | |

逐步添加规则，有选择地允许流量在两者之间流动 namespaces/pods

了解应用程序要求并根据需要创建细粒度的入口和出口规则。以下示例显示了如何将端口 80 上的入口流量限制为 fro app-one m client-one。这有助于最大限度地减少攻击面并降低未经授权访问的风险。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-app-one
  namespace: default
spec:
  podSelector:
    matchLabels:
      k8s-app: app-one
  policyTypes:
  - Ingress
  ingress:
  - from:
```

```
    - podSelector:
        matchLabels:
          k8s-app: client-one
    ports:
    - protocol: TCP
      port: 80
```

允许进入 app-one

| | | Target | | Source | | | | |
|---|---|---|---|---|---|---|---|---|
| # | Policy name | Namespace | Pods | Namespace | Pods | Subnet | Ports | |
| 1 | allow-ingress-app-one | default | k8s-app: app-one | default | k8s-app: client-one | | TCP: 80 | |

## 监控网络策略的执行

- 使用网络策略编辑器
  - 网络策略编辑器可帮助从网络流日志中自动生成可视化、安全评分
  - 以交互方式制定网络策略
- 审核日志
  - 定期查看 EKS 集群的审计日志
  - 审计日志提供了有关在集群上执行了哪些操作的大量信息，包括对网络策略的更改
  - 使用此信息跟踪网络策略随时间推移而发生的变化，并检测任何未经授权或意外的更改
- 自动测试
  - 通过创建镜像您的生产环境的测试环境来实现自动化测试，并定期部署试图违反您的网络策略的工作负载。
- 监控指标
  - 配置您的可观察性代理以从 VPC CNI 节点代理中获取 prometheus 指标，从而监控代理运行状况和 sdk 错误。
- 定期审核网络策略
  - 定期审核您的网络策略，确保它们符合您当前的应用程序要求。随着应用程序的发展，审计使您有机会删除多余的入口和出口规则，并确保您的应用程序没有过多的权限。
- 使用开放策略代理 (OPA) 确保存在网络策略
  - 使用如下所示的 OPA 策略，确保在加载应用程序 pod 之前始终存在网络策略。`k8s-app: sample-app`如果不存在相应的网络策略，则此政策将拒绝使用标签加载 k8s pod。

```
package kubernetes.admission
import data.kubernetes.networkpolicies

deny[msg] {
    input.request.kind.kind == "Pod"
    pod_label_value := {v["k8s-app"] | v := input.request.object.metadata.labels}
    contains_label(pod_label_value, "sample-app")
    np_label_value := {v["k8s-app"] | v :=
 networkpolicies[_].spec.podSelector.matchLabels}
    not contains_label(np_label_value, "sample-app")
    msg:= sprintf("The Pod %v could not be created because it is missing an associated
 Network Policy.", [input.request.object.metadata.name])
}
contains_label(arr, val) {
    arr[_] == val
}
```

## 问题排查

监控 vpc-network-policy-Controller、节点代理日志

启用 EKS 控制平面控制器管理器日志以诊断网络策略功能。您可以将控制平面日志流式传输到 CloudWatch 日志组，并使用CloudWatch日志见解来执行高级查询。从日志中，您可以查看哪些 Pod 端点对象已解析为网络策略、策略的协调状态以及调试策略是否按预期运行。

此外，Amazon VPC CNI 允许您从 EKS 工作节点收集策略执行日志并将其导出到 Amazon Cloudwatch。启用后，您可以利用 Conta CloudWatchiner Insig hts 来提供与网络策略相关的使用情况的见解。

Amazon VPC CNI 还发布了一个软件开发工具包，该软件开发工具包提供了与节点上的 eBPF 程序进行交互的接口。SDK `aws-node` 是在部署到节点上时安装的。你可以在节点的`/opt/cni/bin`目录下找到安装的 SDK 二进制文件。在发布时，SDK为基本功能提供支持，例如检查eBPF程序和地图。

```
sudo /opt/cni/bin/aws-eks-na-cli ebpf progs
```

记录网络流量元数据

AWS VPC Flow Logs 会捕获有关流经 VPC 的流量的元数据，例如源和目标 IP 地址和端口以及 accepted/dropped 数据包。可以对这些信息进行分析，以查找 VPC 内的资源（包括 Pod）之间是否存在可疑或异常活动。但是，由于 Pod 的 IP 地址在被替换时经常发生变化，因此 Flow Logs 本身可能还不够。Calico Enterprise 使用吊舱标签和其他元数据扩展了流日志，从而更容易破译吊舱之间的流量。

# 安全组

EK S 使用 AWS VPC 安全组 (SG) 来控制 Kubernetes 控制平面和集群工作节点之间的流量。安全组还用于控制工作节点与其他 VPC 资源以及外部 IP 地址之间的流量。当您配置 EKS 集群（使用 Kubernetes 版本 1.14-eks.3 或更高版本）时，系统会自动为您创建一个集群安全组。此安全组允许 EKS 控制平面与托管节点组中的节点之间进行不受限制的通信。为简单起见，建议您将集群 SG 添加到所有节点组，包括非托管节点组。

在 Kubernetes 版本 1.14 和 EKS 版本 eks.3 之前，为 EKS 控制平面和节点组配置了单独的安全组。控制平面和节点组安全组的最低和建议规则可在控制平面安全组的最低规则中找到。控制平面安全组的最低规则允许端口 443 从工作节点 SG 入站。https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html.这条规则允许 kubelet 与 Kubernetes API 服务器进行通信。它还包括用于工作节点 SG 的出站流量的端口 10250；10250 是 kubelets 监听的端口。同样，最低节点组规则允许端口 10250 从控制平面 SG 入站，443 出站到控制平面 SG。最后，还有一条规则允许节点组内的节点之间不受限制地进行通信。

如果您需要控制在集群内运行的服务与在集群外运行的服务（例如 RDS 数据库）之间的通信，请考虑为 Pod 设置安全组。借助 Pod 的安全组，您可以将现有安全组分配给一组 Pod。

> **Warning**
>
> 如果您引用的安全组在创建 Pod 之前不存在，则不会调度这些 Pod。

您可以通过创建`SecurityGroupPolicy`对象并指定`PodSelector`或来控制将哪些 Pod 分配给安全组`ServiceAccountSelector`。将选择器设置为`{}`会将中引用的 SG 分配给命名空间中的所有 pod 或命名空间中的所有服务帐户。`SecurityGroupPolicy`在为 pod 实施安全组之前，请务必熟悉所有注意事项。

> **Important**
>
> 如果您将 SG 用于 Pod，则必须创建允许端口 53 出站到集群安全组的 SG。同样，您必须更新集群安全组以接受来自 pod 安全组的端口 53 入站流量。

> **Important**
>
> 对容器使用安全组时，安全组的限制仍然适用，因此请谨慎使用安全组。

> **Important**
>
> 您必须为为 pod 配置的所有探测器为来自集群安全组 (kubelet) 的入站流量创建规则。

> **Important**
>
> Pod 的安全组依赖于名为 ENI 中继的功能，该功能旨在提高 EC2 实例的 ENI 密度。将容器分配给 SG 时，VPC 控制器会将节点组中的分支 ENI 与该容器关联起来。如果在调度 Pod 时，节点组中没有足够的分支 ENI 可用，则该 Pod 将保持待处理状态。实例可以支持的分支 ENI 数量因实例 type/family而异。有关更多详细信息，请参阅 https://docs.aws.amazon.com/eks/latest/userguide/security-groups-for-pods.html #supported-instance-types。

虽然 pod 的安全组提供了 AWS-native 一种无需策略守护程序开销即可控制集群内外的网络流量的方法，但还有其他选项可供选择。例如，Cilium 策略引擎允许您在网络策略中引用 DNS 名称。Calico Enterprise 包括一个用于将网络策略映射到 AWS 安全组的选项。如果您已经实现了像 Istio 这样的服务网格，则可以使用出口网关将网络出口限制为特定的、完全限定的域或 IP 地址。有关此选项的更多信息，请阅读 Istio 中关于出口流量控制的三部分系列。

## 何时对 Pod 使用网络策略与安全组？

### 何时使用 Kubernetes 网络策略

- 控制 pod 到 Pod 的流量
  - 适用于控制集群内 Pod 之间的网络流量（东西向流量）
- 在 IP 地址或端口级别（OSI 第 3 层或第 4 层）控制流量

### 何时使用适用于 Pod 的 AWS 安全组 (SGP)

- 利用现有的 AWS 配置
  - 如果您已经有一组复杂的 EC2 安全组来管理 AWS 服务的访问权限，并且您正在将应用程序从 EC2 实例迁移到 EKS，那么 SGP 可能是一个不错的选择，允许您重复使用安全组资源并将其应用于您的 pod。
- 控制对 AWS 服务的访问权限

  - 在 EKS 集群中运行的应用程序希望与其他 AWS 服务（RDS 数据库）通信，使用 SGP 作为控制从 pod 到 AWS 服务的流量的有效机制。
- 隔离 Pod 和节点流量
  - 如果要将 Pod 流量与其余节点流量完全分开，请在`POD_SECURITY_GROUP_ENFORCING_MODE=strict`模式下使用 SGP。

## 为 Pod 使用安全组和网络策略的最佳实践

- 分层安全
  - 将 SGP 和 kubernetes 网络策略的组合用于分层安全方法
  - 使用 SGP 限制对不属于集群的 AWS 服务的网络级访问，而 kubernetes 网络策略可以限制集群内 Pod 之间的网络流量
- 最低特权原则
  - 只允许 pod 或命名空间之间必要的流量
- 对您的应用程序进行细分
  - 尽可能根据网络策略对应用程序进行分段，以便在应用程序受到威胁时缩小爆炸半径
- 保持政策简单明了
  - Kubernetes 网络策略可能非常精细和复杂，最好让它们尽可能简单，以降低配置错误的风险并减轻管理开销
- 缩小攻击面
  - 通过限制应用程序的暴露来最大限度地减少攻击面

> **Important**
>
> Pod 的安全组提供两种强制模式：`strict`和`standard`。在 EKS 集群中同时使用网络策略和 Pod 安全组功能时，必须使用`standard`模式。

在网络安全方面，分层方法通常是最有效的解决方案。结合使用 kubernetes 网络策略和 SGP 可以为在 EKS 中运行的应用程序提供强大的深度防御策略。

# 服务网格策略实施或 Kubernetes 网络策略

A `service mesh` 是一个专用的基础架构层，您可以将其添加到应用程序中。它允许您透明地添加可观察性、流量管理和安全性等功能，而无需将其添加到自己的代码中。

服务网格在 OSI 模型的第 7 层（应用程序）强制执行策略，而 kubernetes 网络策略在第 3 层（网络）和第 4 层（传输）上运行。这个领域有许多产品，比如 AWS AppMesh、Istio、Linkerd 等，

## 何时使用服务网格执行策略

- 对服务网格进行现有投资
- 需要更高级的功能，例如流量管理、可观察性和安全性
  - 流量控制、负载平衡、断路、速率限制、超时等
  - 详细了解您的服务表现（延迟、错误率、每秒请求数、请求量等）
  - 你想实现和利用服务网格来实现 mTLS 等安全功能

## 选择 Kubernetes 网络策略以获得更简单的用例

- 限制哪些 pod 可以相互通信
- 与服务网格相比，网络策略需要的资源更少，因此非常适合更简单的用例或较小的集群，在这些群集中，运行和管理服务网格的开销可能不合理

> **Note**
>
> 网络策略和服务网格也可以一起使用。使用网络策略在 Pod 之间提供基本的安全性和隔离，然后使用服务网格添加其他功能，例如流量管理、可观察性和安全性。

# ThirdParty 网络策略引擎

如果您有高级策略要求，例如全球网络策略、支持基于DNS主机名的规则、第7层规则、基于规则和明确的 deny/log 操作等， ServiceAccount 可以考虑使用第三方网络策略引擎。Calico 是 Tiger a的开源策略引擎，可以很好地与EKS配合使用。除了实现全套 Kubernetes 网络策略功能外，Calico 还支持具有更丰富功能的扩展网络策略，包括与 Istio 集成时对第 7 层规则（例如 HTTP）的支持。Calico 策略的范围可以限定为命名空间、Pod、服务帐号或全局。当策略的范围限定为服务帐号时，它会将一组 ingress/egress 规则与该服务帐号相关联。有了适当的 RBAC 规则，您就可以防止团队推翻这些

规则，从而允许 IT 安全专业人员安全地委托对命名空间的管理。Cilium 的维护者Isovalent还将网络策略扩展到包括对第7层规则（例如HTTP）的部分支持。Cilium 还支持 DNS 主机名，这可用于限制 Kubernetes 与在您的 VPC 内外运行的资源 Services/Pods 之间的流量。相比之下，Calico Enterprise 包含一项功能，允许你将 Kubernetes 网络策略映射到 AWS 安全组以及 DNS 主机名。

你可以在以下网址找到常见的 Kubernetes 网络策略列表。Calico https://github.com/ahmetb/kubernetes-network-policy-recipes. 的类似规则集可在以下网址找到 https://docs.projectcalico.org/security/calico-network-policy.

## 迁移到 Amazon VPC CNI 网络策略引擎

为了保持一致性并避免意外的 pod 通信行为，建议在集群中仅部署一个网络策略引擎。如果您想从 3P 迁移到 VPC CNI 网络策略引擎，我们建议您在启用 VPC CNI 网络策略支持之前将现有的 3P NetworkPolicy CRD 转换为 Kubernetes NetworkPolicy 资源。而且，在生产环境中应用迁移的策略之前，在单独的测试集群中对其进行测试。这使您可以识别和解决 Pod 通信行为中的任何潜在问题或不一致之处。

迁移工具

为了帮助您完成迁移过程，我们开发了一种名为 K8s 网络策略迁移器的工具，可将您现有的网络策略 CRD 转换为 Kubernetes 原生 Calico/Cilium 网络策略。转换后，您可以直接在运行 VPC CNI 网络策略控制器的新集群上测试转换后的网络策略。该工具旨在帮助您简化迁移过程并确保平稳过渡。

> **Important**
>
> 迁移工具只会转换与原生 kubernetes 网络策略 api 兼容的 3P 策略。如果您使用的是3P插件提供的高级网络策略功能，则迁移工具将跳过并报告它们。

请注意，AWS VPC CNI 网络策略工程团队目前不支持迁移工具，而是尽最大努力向客户提供该工具。我们鼓励您使用此工具来简化您的迁移过程。如果您在使用该工具时遇到任何问题或错误，我们恳请您创建GitHub问题。您的反馈对我们来说非常宝贵，将有助于我们不断改进服务。

## 其他资源

- Kubernetes & Tigera：网络策略、安全和审计
- 印花布企业
- Cilium

- NetworkPolicy编辑 Cilium 的交互式策略编辑器
- Inspektor Gadget 建议网络策略小工具根据对网络流量的分析建议网络策略

## 传输中加密

需要符合 PCI、HIPAA 或其他法规的应用程序可能需要在传输数据时对其进行加密。如今，TLS已成为加密网络流量的事实选择。TLS 与其前身 SSL 一样，使用加密协议通过网络提供安全的通信。TLS 使用对称加密，其中加密数据的密钥是根据会话开始时协商的共享密钥生成的。以下是您可以在 Kubernetes 环境中加密数据的几种方法。

### Nitro 实例

默认情况下，以下 Nitro 实例类型（例如 c5n、G4、i3en、m5dn、m5dn、m5n、p3dn、r5dn 和 r5n）之间交换的流量会自动加密。当有中间跳点（例如传输网关或负载均衡器）时，流量不会被加密。有关传输中加密的更多详细信息以及默认情况下支持网络加密的实例类型的完整列表，请参阅传输中的加密。

### 服务网格

传输中的加密也可以使用 App Mesh、Linkerd v2 和 Istio 等服务网格来实现。 AppMesh 支持带有 X.509 证书的 mTLS 或 Envoy 的秘密发现服务 (SDS)。Linkerd 和 Istio 都支持 mTLS。

aws-app-mesh-examples GitHub 存储库提供了使用 X.509证书配置 mTLS 和 SPIRE 作为你的 Envoy 容器的 SDS 提供者的演练：

- 使用证书配置 mTLS X.509
- 使用 SPIRE (SDS) 配置 TLS

App Mesh 还支持使用 AWS Certificate Manager (ACM) 颁发的私有证书或存储在虚拟节点本地文件系统中的证书进行 TLS 加密。

aws-app-mesh-examples GitHub 存储库提供了使用 ACM 颁发的证书和随你的 Envoy 容器打包的证书配置 TLS 的演练：

- 使用文件提供的 TLS 证书配置 TLS
- 使用 AWS Certifice Manager 配置 TLS

## 入口控制器和负载均衡器

入口控制器是您智能地将来自集群外部的 HTTP/S流量路由到集群内部运行的服务的一种方式。通常，这些 Ingress 前面有第 4 层负载均衡器，例如 Classic Load Balancer 或 Network Load Balancer (NLB)。加密流量可以在网络中的不同位置终止，例如在负载均衡器、入口资源或 Pod 上。终止 SSL 连接的方式和地点最终将由贵组织的网络安全策略决定。例如，如果您的策略要求端到端加密，则必须解密 Pod 上的流量。这将给你的 Pod 带来额外的负担，因为它必须花费周期来建立初始握手。总体 SSL/TLS 处理占用大量的 CPU 资源。因此，如果你有灵活性，可以尝试在 Ingress 或负载均衡器上执行 SSL 卸载。

在 AWS 弹性负载均衡器中使用加密

AWS Applicat ion Load Bal ancer (ALB) 和网络负载均衡器 (NLB) 都支持传输加密（SSL 和 TLS）。ALB 的`alb.ingress.kubernetes.io/certificate-arn`注释允许您指定要向 ALB 添加哪些证书。如果您省略注释，控制器将尝试使用主机字段匹配可用的 AWS Certificate Manager (ACM) 证书，从而向需要该注释的侦听器添加证书。从 EKS v1.15 开始，你可以将`service.beta.kubernetes.io/aws-load-balancer-ssl-cert`注释与 NLB 一起使用，如下例所示。

```
apiVersion: v1
kind: Service
metadata:
  name: demo-app
  namespace: default
  labels:
    app: demo-app
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "<certificate ARN>"
     service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http"
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 80
    protocol: TCP
  selector:
    app: demo-app
//---
kind: Deployment
apiVersion: apps/v1
```

```
metadata:
  name: nginx
  namespace: default
  labels:
    app: demo-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-app
  template:
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 443
              protocol: TCP
            - containerPort: 80
              protocol: TCP
```

以下是其他 SSL/TLS 终止示例。

- 使用 Contour 保护 EKS 入口然后我们来加密吧 GitOps
- 如何使用 ACM 终止亚马逊 EKS 工作负载上的 HTTPS 流量？

> **Important**
>
> 某些 Ingress，比如 AWS LB 控制器， SSL/TLS 使用注解来实现，而不是作为入口规范的一部分。

### 带证书管理器的 ACM 私有 CA

您可以使用 ACM 私有证书颁发机构 (CA) 和证书管理器（一种流行的 Kubernetes 附加组件，用于分发、续订和吊销证书）启用 TLS 和 m TLS，以保护入口、容器上和 Pod 之间的 EKS 应用程序工作负载。ACM Private CA 是一种高度可用、安全、托管的 CA，无需管理自己的 CA 所需的前期成本和维护成本。如果您使用的是默认 Kubernetes 证书颁发机构，则有机会提高安全性并满足 ACM 私有 CA

的合规性要求。与存储在内存中编码的密钥的默认 CA（不太安全）相比，ACM Private CA 在 FIPS 140-2 3 级硬件安全模块中保护私钥（非常安全）。集中式 CA 还可让您更好地控制 Kubernetes 环境内部和外部的私有证书，并提高其可审计性。

Short-Lived 用于工作负载之间双向 TLS 的 CA 模式

在 EKS 中为 mTLS 使用 ACM 私有 CA 时，建议您在短期 CA 模式下使用短期证书。尽管可以在通用 CA 模式下签发短期证书，但对于需要经常颁发新证书的用例，使用短期 CA 模式更具成本效益（比普通模式便宜约 75%）。除此之外，您还应尝试将私有证书的有效期与 EKS 集群中 pod 的生命周期保持一致。在此处@@ 了解有关 ACM 私有 CA 及其优势的更多信息。

ACM 安装说明

首先按照 ACM 私有 CA 技术文档中提供的步骤创建私有 CA。拥有私有 CA 后，请按照常规安装说明安装证书管理器。安装证书管理器后，按照中的设置说明安装私有 CA Kubernetes 证书管理器插件。GitHub该插件允许证书管理器向 ACM 私有 CA 请求私有证书。

现在你已经有了私有 CA 和一个安装了证书管理器和插件的 EKS 集群，是时候设置权限并创建颁发者了。更新 EKS 节点角色的 IAM 权限以允许访问 ACM 私有 CA。用您的私<CA_ARN>有 CA 中的值替换：

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "awspcaissuer",
            "Action": [
                "acm-pca:DescribeCertificateAuthority",
                "acm-pca:GetCertificate",
                "acm-pca:IssueCertificate"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:acm-pca:us-west-2:123456789012:certificate-
authority/12345678-1234-1234-1234-123456789012"
        }
    ]
}
```

也可以使用 IAM 账户或 IRSA 的服务角色。有关完整示例，请参阅下面的 “其他资源” 部分。

在 Amazon EKS 中创建名为 cluster-issuer.yaml 的自定义资源定义文件，其中包含以下文本，用您的私有 CA 替换<CA_ARN>和<Region>信息。

```
apiVersion: awspca.cert-manager.io/v1beta1
kind: AWSPCAClusterIssuer
metadata:
          name: demo-test-root-ca
spec:
          arn: <CA_ARN>
          region: <Region>
```

部署您创建的发行者。

```
kubectl apply -f cluster-issuer.yaml
```

您的 EKS 集群已配置为向私有 CA 请求证书。现在，您可以使用 cert-manager 的`Certificate`资源来颁发证书，方法是将该`issuerRef`字段的值更改为您在上面创建的私有 CA 颁发者。有关如何指定和请求证书资源的更多详细信息，请查看 cert-manager 的证书资源指南。请@@ 在此处查看示例。

## 带有 Istio 和证书管理器的 ACM 私有 CA

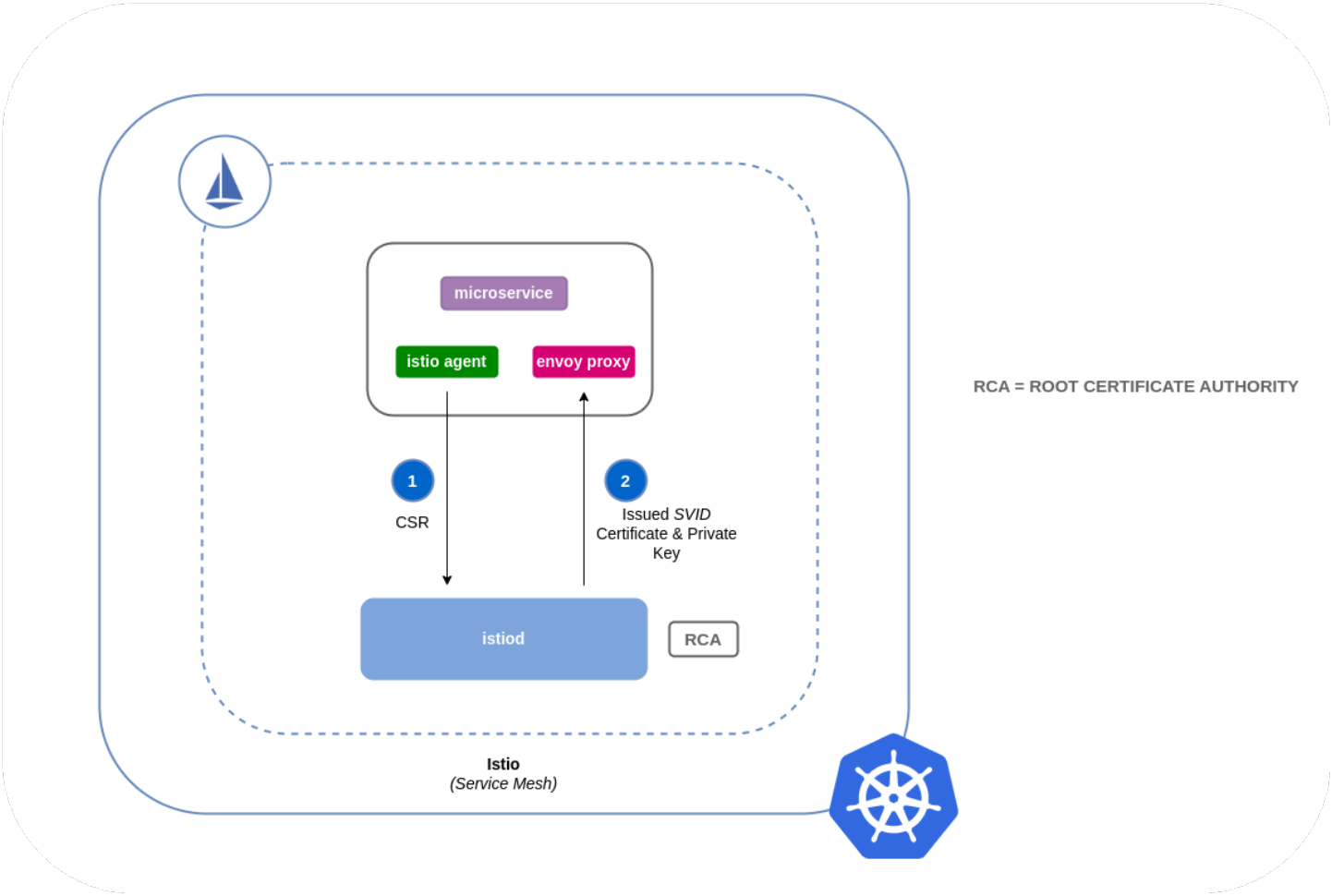
如果您在 EKS 集群中运行 Istio，则可以禁用 Istio 控制平面（特别是`istiod`）作为根证书颁发机构 (CA) 的功能，并将 ACM 私有 CA 配置为工作负载之间 mTLS 的根 CA。如果你要使用这种方法，可以考虑在 ACM 私有 CA 中使用短暂的 CA 模式。有关更多详细信息，请参阅上一节和此博客文章。

证书签名在 Istio 中的工作原理（默认）

Kubernetes 中的工作负载是使用服务帐户识别的。如果您未指定服务帐户，Kubernetes 将自动为您的工作负载分配一个服务帐户。此外，服务帐号会自动挂载关联的令牌。该令牌由服务账户用于工作负载，以根据 Kubernetes API 进行身份验证。服务账号可能足以作为 Kubernetes 的身份，但是 Istio 有自己的身份管理系统和 CA。当工作负载使用其 envoy sidecar 代理启动时，它需要从 Istio 分配一个身份才能被视为值得信赖并允许与网格中的其他服务进行通信。

为了从 Istio 获取此身份，会向 Istio 控制平面`istio-agent`发送一个名为证书签名请求（或 CSR）的请求。此 CSR 包含服务帐户令牌，因此可以在处理工作负载之前验证其身份。此验证过程由`istiod`同时充当注册机构（或 RA）和 CA 的机构处理。RA 充当看门人，确保只有经过验证的 CSR 才能进入 CA。CSR 通过验证后，将转发给 CA，然后由证书颁发包含服务账户的 SPIFFE 身份的证书。此证书被称为 SPIFFE 可验证身份证件（或 SVID）。SVID 被分配给请求服务，用于识别和加密通信服务之间传输的流量。

Istio 证书签名请求的默认流程：




在 Istio 中如何使用 ACM 私有 CA 进行证书签名

你可以使用名为 Istio 证书签名请求代理 (istio-csr) 的证书管理器插件将 Ist io 与 ACM 私有 CA 集成。该代理允许证书管理器颁发者（在本例中为 ACM Private CA）保护 Istio 工作负载和控制平面组件。istio-csr 代理公开的服务与 i stiod 在验证传入的 CSR 的默认配置中提供的服务相同。但是，在验证后，它会将请求转换为证书管理器支持的资源（即与外部 CA 颁发机构的集成）。

每当工作负载有 CSR 时，它就会被转发给 istio-csr，后者将向 ACM Private CA 申请证书。istio-csr 和 ACM 私有 CA 之间的这种通信由 AW S 私有 CA 颁发者插件启用。证书管理器使用此插件向 ACM 私有 CA 请求 TLS 证书。颁发者插件将与 ACM 私有 CA 服务通信，请求工作负载的签名证书。证书签名后，它将返回给 istio-csr，后者将读取已签名的请求，并将其返回到启动 CSR 的工作负载。

使用 istio-csr 提交 Istio 证书签名请求流程

图片:: istio-csr-with-acm-private-ca.png [使用 istio-csr 进行 Istio 证书签名请求的流程]

带有私有 CA 设置说明的 Istio

1. 首先按照本节中的相同设置说明完成以下操作：
2. 创建私有 CA
3. 安装证书管理器
4. 安装发行者插件
5. 设置权限并创建颁发者。颁发者代表 CA，用于签署istiod和网格化工作负载证书。它将与 ACM 私有 CA 通信。
6. 创建istio-system命名空间。这是 Istio istiod certificate 和其他 Istio 资源将部署到此处。
7. 安装配置有 AWS 私有 CA 发行者插件的 Istio CSR。您可以保留工作负载的证书签名请求，以验证它们是否获得批准和签名 (preserveCertificateRequests=true)。

```
helm install -n cert-manager cert-manager-istio-csr jetstack/cert-manager-istio-csr \
--set "app.certmanager.issuer.group=awspca.cert-manager.io" \
--set "app.certmanager.issuer.kind=AWSPCAClusterIssuer" \
--set "app.certmanager.issuer.name=<the-name-of-the-issuer-you-created>" \
--set "app.certmanager.preserveCertificateRequests=true" \
--set "app.server.maxCertificateDuration=48h" \
--set "app.tls.certificateDuration=24h" \
--set "app.tls.istiodCertificateDuration=24h" \
--set "app.tls.rootCAFile=/var/run/secrets/istio-csr/ca.pem" \
--set "volumeMounts[0].name=root-ca" \
--set "volumeMounts[0].mountPath=/var/run/secrets/istio-csr" \
--set "volumes[0].name=root-ca" \
--set "volumes[0].secret.secretName=istio-root-ca"
```

8. 使用自定义配置安装 Istio，cert-manager istio-csr将其istiod替换为网格的证书提供商。这个过程可以使用 Istio 运算符来执行。

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio
  namespace: istio-system
spec:
  profile: "demo"
  hub: gcr.io/istio-release
  values:
  global:
```

```
    # Change certificate provider to cert-manager istio agent for istio agent
    caAddress: cert-manager-istio-csr.cert-manager.svc:443
  components:
    pilot:
      k8s:
        env:
          # Disable istiod CA Sever functionality
        - name: ENABLE_CA_SERVER
          value: "false"
        overlays:
        - apiVersion: apps/v1
          kind: Deployment
          name: istiod
          patches:

            # Mount istiod serving and webhook certificate from Secret mount
          - path: spec.template.spec.containers.[name:discovery].args[7]
            value: "--tlsCertFile=/etc/cert-manager/tls/tls.crt"
          - path: spec.template.spec.containers.[name:discovery].args[8]
            value: "--tlsKeyFile=/etc/cert-manager/tls/tls.key"
          - path: spec.template.spec.containers.[name:discovery].args[9]
            value: "--caCertFile=/etc/cert-manager/ca/root-cert.pem"

          - path: spec.template.spec.containers.[name:discovery].volumeMounts[6]
            value:
              name: cert-manager
              mountPath: "/etc/cert-manager/tls"
              readOnly: true
          - path: spec.template.spec.containers.[name:discovery].volumeMounts[7]
            value:
              name: ca-root-cert
              mountPath: "/etc/cert-manager/ca"
              readOnly: true

          - path: spec.template.spec.volumes[6]
            value:
              name: cert-manager
              secret:
                secretName: istiod-tls
          - path: spec.template.spec.volumes[7]
            value:
              name: ca-root-cert
              configMap:
                defaultMode: 420
```

```
                name: istio-ca-root-cert
```

9. 部署您创建的上述自定义资源。

```
istioctl operator init
kubectl apply -f istio-custom-config.yaml
```

10现在，您可以将工作负载部署到 EKS 集群中的网格并强制执行 mTLS。

Istio 证书签名请求

图片:: istio-csr-requests.png [Istio 证书签名请求]

## 工具和资源

- Amazon EKS 安全沉浸式研讨会-网络安全
- 如何实现证书管理器和 ACM 私有 CA 插件以在 EKS 中启用 TLS。
- 使用@@ 新的 AWS Load Balancer 控制器和 ACM 私有 CA 在 Amazon EKS 上设置端到端 TLS 加密。
- 私有 CA Kubernetes 证书管理器插件已开启。 GitHub
- 私有 CA Kubernetes 证书管理器插件用户指南。
- 如何使用 AWS 私有证书颁发机构短期证书模式
- egress-operator 一个操作员和 DNS 插件，无需协议检查即可控制来自集群的出口流量
- NeuVector 由 SUSE 开源提供的零信任容器安全平台提供策略网络规则、数据丢失防护 (DLP)、Web 应用程序防火墙 (WAF) 和网络威胁签名。

# 数据加密和机密管理

## 静态加密

Kubernetes 有三种不同的 AWS-native 存储选项可供使用：E BS、EFS 和 FSx for Lustre。所有三者都使用服务托管式密钥或客户主密钥（CMK）进行静态加密。对于 EBS，您可以使用树内存储驱动程序或 EBS CSI 驱动程序。两者都包含用于加密卷和提供 CMK 的参数。对于 EFS，您可以使用 EFS CSI 驱动程序，但是，与 EBS 不同，EFS CSI 驱动程序不支持动态配置。如果要将 EFS 与 EKS 配合使用，则需要在创建 PV 之前为文件系统配置和配置静态加密。有关 EFS 文件加密的更多信息，请参阅加密静态数据。除了提供静态加密外，EFS 和 FSx for Lustre 还包括对传输中的数据进行加密

的选项。默认情况下，FSx for Lustre 会这样做。对于 EFS，您可以通过在 PV `mountOptions` 中添加`tls`参数来添加传输加密，如以下示例所示：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: efs-sc
  mountOptions:
    - tls
  csi:
    driver: efs.csi.aws.com
    volumeHandle: <file_system_id>
```

FSx CSI 驱动程序支持 Lustre 文件系统的动态配置。默认情况下，它使用服务托管密钥对数据进行加密，但可以选择提供您自己的 CMK，如以下示例所示：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fsx-sc
provisioner: fsx.csi.aws.com
parameters:
  subnetId: subnet-056da83524edbe641
  securityGroupIds: sg-086f61ea73388fb6b
  deploymentType: PERSISTENT_1
  kmsKeyId: <kms_arn>
```

> **Important**
>
> 自 2020 年 5 月 28 日起，写入 EKS Fargate 容器中临时卷的所有数据都默认使用行业标准的加密算法进行加密。 AES-256 由于加密和解密由该服务无缝处理，因此无需对应用程序进行任何修改。

### 加密静态数据

加密静态数据被认为是一种最佳实践。如果您不确定是否需要加密，请加密您的数据。

### 定期轮换您的 CMK

将 KMS 配置为自动轮换您的 CMK。这将每年轮换一次您的密钥，同时无限期地保存旧密钥，这样您的数据仍然可以被解密。有关更多信息，请参阅轮换客户主密钥

### 使用 EFS 接入点简化对共享数据集的访问

如果您共享了具有不同 POSIX 文件权限的数据集，或者想要通过创建不同的挂载点来限制对部分共享文件系统的访问，请考虑使用 EFS 接入点。要了解有关使用接入点的更多信息，请参阅 T https://docs.aws.amazon.com/efs/latest/ug/efs-access-points.html. oday，如果要使用接入点 (AP)，则需要在 PV 的`volumeHandle`参数中引用 AP。

> **Important**
>
> 自 2021 年 3 月 23 日起，EFS CSI 驱动程序支持动态配置 EFS 接入点。接入点是 EFS 文件系统中特定于应用程序的入口点，可以更轻松地在多个 pod 之间共享文件系统。每个 EFS 文件系统最多可以有 120 个 PV。有关更多信息，请参阅 Amazon EFS CSI 动态配置简介。

## 密钥管理

Kubernetes 密钥用于存储敏感信息，例如用户证书、密码或 API 密钥。它们作为 base64 编码的字符串保存在 etcd 中。在 EKS 上，etcd 节点的 EBS 卷使用 E BS 加密进行加密。Pod 可以通过引用中的密钥来检索 Kubernetes 机密对象。`podSpec`这些密钥可以映射到环境变量，也可以作为卷挂载。有关创建密钥的更多信息，请参阅 https://kubernetes.io/docs/concepts/configuration/secret/.

> **Warning**
>
> 特定命名空间中的密钥可以被该密钥命名空间中的所有 pod 引用。

> **Warning**
>
> 节点授权器允许 Kubelet 读取挂载到节点上的所有密钥。

## 使用 AWS KMS 对 Kubernetes 机密进行信封加密

这允许您使用唯一的数据加密密钥 (DEK) 加密您的机密。然后，使用来自 AWS KMS 的密钥加密密钥 (KEK) 对 DEK 进行加密，该密钥可以定期自动轮换。使用适用于 Kubernetes 的 KMS 插件，所有 Kubernetes 机密都以密文而不是纯文本形式存储在 etcd 中，并且只能由 Kubernetes API 服务器解密。有关更多详细信息，请参阅使用 EKS 加密提供商支持进行深入防御

## 审计 Kubernetes 密钥的使用情况

在 EKS 上，打开审核日志并创建 CloudWatch 指标筛选器和警报，以便在使用密钥时提醒您（可选）。以下是 Kubernetes 审计日志的指标筛选器的示例。`{($.verb="get") && ($.objectRef.resource="secret")}`您还可以在 CloudWatch Log Insights 中使用以下查询：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| stats count(*) by objectRef.name as secret
| filter verb="get" and objectRef.resource="secrets"
```

上面的查询将显示在特定时间范围内访问密钥的次数。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter verb="get" and objectRef.resource="secrets"
| display objectRef.namespace, objectRef.name, user.username, responseStatus.code
```

此查询将显示密钥，以及尝试访问该密钥的用户的命名空间和用户名以及响应代码。

## 定期轮换你的秘密

Kubernetes 不会自动轮换密钥。如果您必须轮换密钥，请考虑使用外部密钥存储库，例如 Vault 或 AWS Secrets Manager。

## 使用单独的命名空间来隔离来自不同应用程序的机密

如果您的密钥无法在命名空间中的应用程序之间共享，请为这些应用程序创建一个单独的命名空间。

## 使用卷装载而不是环境变量

环境变量的值可能会无意中出现在日志中。作为卷挂载的密钥会被实例化为 tmpfs 卷（支持 RAM 的文件系统），这些卷将在删除 Pod 时自动从节点中删除。

## 使用外部密钥提供者

除了使用 Kubernetes 机密之外，还有几种可行的替代方案，包括 AWS Secrets Manager 和 Hashicorp 的 Vault。这些服务提供了 Kubernetes Secrets 无法提供的精细访问控制、强加密和自动轮换机密等功能。Bitnami的密封机密是另一种使用非对称加密来创建 “密封机密” 的方法。公钥用于加密密钥，而用于解密密密钥的私钥则保存在集群中，这样您就可以将密封的机密安全地存储在 Git 等源代码控制系统中。有关更多信息，请参阅使用密封密钥在 Kubernetes 中管理密钥部署。

随着外部密钥存储库的使用越来越多，将它们与 Kubernetes 集成的需求也随之增加。S ecret Store CSI 驱动程序是一个社区项目，它使用 CSI 驱动程序模型从外部秘密存储库中获取机密。目前，该驱动程序支持 AWS Secrets Manager、Azure、Vault 和 GCP。AWS 提供商同时支持 AWS Secrets Manager 和 AWS Parameter Store。也可以将其配置为在密钥过期时轮换，并且可以将 AWS Secrets Manager 密钥同步到 Kubernetes 密钥。当您需要将密钥作为环境变量引用而不是从卷中读取密钥时，同步密钥会很有用。

> **Note**
>
> 当密钥存储库 CSI 驱动程序必须获取密钥时，它会假设分配给引用密钥的 pod 的 IRSA 角色。此操作的代码可以在这里找到。

有关 AWS 机密和配置提供商 (ASCP) 的更多信息，请参阅以下资源：

- 如何将 AWS 密钥配置提供程序与 Kubernetes 密钥存储 CSI 驱动程序配合使用
- 将 Secrets Manager 机密与 Kubernetes 密钥商店 CSI 驱动程序集成

外部机密是在 Kubernetes 中使用外部密钥存储库的另一种方式。与 CSI 驱动程序一样，外部机密适用于各种不同的后端，包括 AWS Secrets Manager。不同之处在于，外部机密不是从外部密钥存储中检索机密，而是将这些后端的机密作为机密复制到 Kubernetes。这使您可以使用首选的密钥存储库来管理密钥，并以 Kubernetes-native 某种方式与密钥进行交互。

## 工具和资源

- Amazon EKS 安全沉浸式研讨会-数据加密和机密管理

# 运行时系统安全性

运行时安全性为您的容器在运行时提供主动保护。其目的是检测， and/or 防止容器内发生恶意活动。这可以通过 Linux 内核中的许多机制或与 Kubernetes 集成的内核扩展来实现，例如 Linux 功能、安全计算 (seccomp) 或 SELinux。 AppArmor还有诸如Amazon GuardDuty 和第三方工具之类的选项，可以帮助建立基准和检测异常活动，而无需手动配置Linux内核机制。

> **Important**
>
> Kubernetes 目前没有提供任何原生机制来将 seccomp、或 SELinux 配置文件加载到 AppArmor节点上。它们要么必须手动加载，要么在启动时安装到节点上。这必须在你的 Pod 中引用它们之前完成，因为调度器不知道哪些节点有配置文件。参见下文，诸如安全配置文件操作员之类的工具如何帮助将配置文件自动配置到节点。

## 安全上下文和内置 Kubernetes 控件

许多 Linux 运行时安全机制都与 Kubernetes 紧密集成，可以通过 Kubernetes 安全上下文进行配置。其中一个选项是`privileged`标志，`false`默认情况下，该标志处于启用状态，则基本上等同于主机上的 root。在生产工作负载中启用特权模式几乎总是不合适的，但是还有更多的控件可以根据需要为容器提供更精细的权限。

### Linux 功能

Linux 功能允许您向 Pod 或容器授予某些功能，而无需提供 root 用户的所有能力。示例包括`CAP_NET_ADMIN`，它允许配置网络接口或防火墙`CAP_SYS_TIME`，或者允许操纵系统时钟。

### Seccomp

使用安全计算 (seccomp)，您可以防止容器化应用程序对底层主机操作系统的内核进行某些系统调用。虽然 Linux 操作系统有几百个系统调用，但其中大部分并不是运行容器所必需的。通过限制容器可以进行的系统调用，可以有效地减少应用程序的攻击面。

Seccomp 的工作原理是拦截系统调用，并且只允许那些被列入许可名单的系统调用通过。Docker 有一个默认的 seccomp 配置文件，适用于大多数通用工作负载，而其他容器运行时（例如 containerd）则提供了类似的默认值。你可以在 Pod 规范的`securityContext`部分中添加以下内容，将你的容器或 Pod 配置为使用容器运行时的默认 seccomp 配置文件：

```
securityContext:
```

```
  seccompProfile:
    type: RuntimeDefault
```

从 1.22 开始（在 alpha 版本中，从 1.27 起就稳定了），使用单个 kubel et 标志，RuntimeDefault可以将上述内容用于节点上的所有 Pod。--seccomp-default然后，只有其他配置文件securityContext才需要中指定的配置文件。

也可以为需要额外权限的内容创建自己的配置文件。手动执行此操作可能非常繁琐，但是有一些工具，例如 Inspektor Gadget（在网络安全部分也推荐用于生成网络策略）和安全配置文件操作员，它们支持使用诸如eBPF或日志之类的工具将基准权限要求记录为seccomp配置文件。Security Profiles Operator 还允许自动将录制的配置文件部署到节点，以供 Pod 和容器使用。

### AppArmor 还有 selinux

AppArmor 而 SELinux 则被称为强制访问控制或 MAC 系统。它们的概念与 seccomp 相似，但具有不同的 API 和功能，允许对特定的文件系统路径或网络端口等进行访问控制。对这些工具的支持取决于 Linux 发行版，其 Debian/Ubuntu 支持 AppArmor 和 L RHEL/CentOS/Bottlerocket/Amazon inux 2023 都支持 SELinux。有关 SELinux 的进一步讨论，另请参阅基础设施安全部分。

两者 AppArmor 和 SELinux 都与 Kubernetes 集成，但是从 Kubernetes 1.28 开始， AppArmor 配置文件必须通过注释指定，而 SELinux 标签可以通过安全上下文中的字段直接设置。SELinuxOptions

与 seccomp 配置文件一样，上面提到的安全配置文件操作员可以帮助将配置文件部署到集群中的节点上。（将来，该项目还旨在像为seccomp一样为 AppArmor 和SELinux生成配置文件。）

## 建议

### 使用 Amazon GuardDuty 进行运行时监控和检测对您的 EKS 环境的威胁

如果您目前没有用于持续监控 EKS 运行时和分析 EKS 审核日志以及扫描恶意软件和其他可疑活动的解决方案，Amazon 强烈建议那些想要简单、快速、安全、可扩展且经济实惠的一键式方法来保护 AWS 环境的客户使用 A ma GuardDuty zon。Amazon GuardDuty 是一项安全监控服务，用于分析和处理基础数据源，例如 AWS CloudTrail 管理事件、AWS CloudTrail 事件日志、VPC 流日志（来自亚马逊 EC2 实例）、Kubernetes 审计日志和 DNS 日志。它还包括 EKS 运行时监控。它使用不断更新的威胁情报源（例如恶意 IP 地址和域名列表）以及机器学习来识别您的 AWS 环境中意外、可能未经授权的恶意活动。这可能包括权限升级、使用暴露的凭证、与恶意 IP 地址、域名通信、Amazon EC2 实例和 EKS 容器工作负载上存在恶意软件或发现可疑 API 活动等问题。 GuardDuty 通过生成可在 GuardDuty 控制台或通过 Amazon EventBridge 查看的安全调查结果来通知您 AWS 环境的状态。GuardDuty 还支持您将调查结果导出到亚马逊简单存储服务 (S3) 存储桶，并与 AWS Security Hub 和 Detective 等其他服务集成。

观看这场 AWS 在线技术讲座“使用亚马逊增强的 Amazon EKS 威胁检测 GuardDuty ——AWS 在线技术讲座”，了解如何在几分钟内逐步启用这些额外的 EKS 安全功能。

### 可选：使用第三方解决方案进行运行时监控

如果你不熟悉 Linux 的安全性，那么创建和管理 seccomp 和 Apparmor 配置文件可能会很困难。如果您没有时间精通，可以考虑使用第三方商业解决方案。他们中的许多人已经超越了Apparmor和seccomp等静态配置文件，并开始使用机器学习来阻止或提醒可疑活动。其中一些解决方案可以在下面的工具部分中找到。其他选项可以在适用于容器的 AWS Marketplace 上找到。

### 在编写 seccomp 策略之前，请考虑 add/dropping Linux 功能

功能包括对可通过 syscalls 访问的内核函数进行各种检查。如果检查失败，系统调用通常会返回错误。检查可以在特定 syscall 的开头完成，也可以在内核的深处，在可能通过多个不同的系统调用（例如写入特定的特权文件）访问的区域中完成。另一方面，Seccomp 是一个系统调用过滤器，在所有系统调用运行之前将其应用于所有系统调用。进程可以设置过滤器，允许他们撤消运行某些系统调用的权限，或者撤消某些系统调用的特定参数的权利。

在使用 seccomp 之前，请考虑 adding/removing Linux 功能是否为你提供了所需的控制权。有关更多信息，请参阅为容器设置功能。

### 看看你能否通过使用 Pod 安全策略 (PSP) 来实现自己的目标

Pod 安全策略提供了许多不同的方法来改善您的安全状况，而不会引入过多的复杂性。在开始构建 seccomp 和 Apparmor 配置文件之前，请先探索 PSP 中可用的选项。

> **Warning**
>
> 从 Kubernetes 1.25 起，PSP 已被移除，取而代之的是容器安全准入控制器。 Third-party 现有的替代方案包括 OPA/Gatekeeper 和 Kyverno。可以从 Gatekeeeper 库存储库中提取一组用于实施 PSP 中常见的策略的 Gatekeeper 约束和约束模板。 GitHub在 Kyverno策略库中可以找到许多PSP的替代品，包括Pod安全标准的完整集合。

## 工具和资源

- 开始之前你应该知道的 7 件事
- AppArmor装载机
- 使用配置文件设置节点

- Sec@@ urity Profiles Operator 是一项 Kubernetes 增强功能，旨在让用户更轻松地在 Kubernetes 集群中使用 SELinux、seccomp 和。 AppArmor 它提供了通过运行的工作负载生成配置文件和将配置文件加载到 Kubernetes 节点以在 Pod 中使用的功能。
- Inspektor Gadget 允许检查、跟踪和分析 Kubernetes 上运行时行为的许多方面，包括协助生成 seccomp 配置文件。
- 水族
- Qualys
- Stackrox
- Sysdig 安全
- Prisma
- NeuVector 由 SUSE 开源提供的零信任容器安全平台提供进程配置文件规则和文件访问规则。

# 保护基础架构（主机）

由于保护您的容器镜像非常重要，因此保护运行这些镜像的基础架构也同样重要。本节探讨降低直接针对主机的攻击所带来的风险的不同方法。这些指南应与 “运行时安全” 部分中概述的指南结合使用。

## 建议

### 使用专为运行容器而优化的操作系统

考虑使用 Flatcar Linux、Project Atomic、RancherOS 和 Bottlerocket，后者是 AWS 专为运行 Linux 容器而设计的特殊用途操作系统。它包括减少攻击面、启动时经过验证的磁盘映像以及使用 SELinux 强制执行的权限边界。

或者，也可以为您的 Kubernetes 工作节点使用 EK S 优化的 AMI。EKS 优化的 AMI 定期发布，其中包含运行容器化工作负载所需的最少一组操作系统包和二进制文件。

请参阅 Amazon EKS AMI RHEL 构建规范，了解示例配置脚本，该脚本可用于使用 Hashicorp Packer 构建在红帽企业 Linux 上运行的自定义 Amazon EKS AMI。可以进一步利用此脚本来构建符合 STIG 标准的 EKS 自定义 AMI。

### 让你的工作节点操作系统保持最新状态

无论您使用的是容器优化的主机操作系统（如 Bottlerocket）还是更大但仍然极简主义的 Amazon Machine Image（如 EKS 优化的 AMI），最佳做法是使用最新的安全补丁使这些主机操作系统映像保持最新状态。

对于 EKS 优化的 AMI，请定期查看 C HANGELOG and/or 发行说明频道，并自动将更新的工作节点映像部署到您的集群中。

### 将您的基础架构视为不可变的，并自动更换工作节点

与其执行就地升级，不如在有新的补丁或更新可用时更换工作人员。有几种方法可以解决这个问题。您可以使用最新的 AMI 向现有自动扩缩组添加实例，因为您可以按顺序封锁和清空节点，直到该组中的所有节点都被最新的 AMI 替换。或者，您可以向新节点组添加实例，同时按顺序封锁和清空旧节点组中的节点，直到所有节点都被替换。EKS 托管节点组使用第一种方法，当新 AMI 可用时，将在控制台中显示一条消息以升级您的工作程序。 `eksctl`还有一种机制，用于使用最新 AMI 创建节点组，并在实例终止之前优雅地封锁和排出节点组中的 Pod。如果您决定使用其他方法来替换工作节点，强烈建议您自动执行该过程以最大限度地减少人为监督，因为随着新 updates/patches 工作节点的发布和控制平面升级，您可能需要定期更换工作节点。

借助 EKS Fargate，AWS 将在更新可用时自动更新底层基础设施。通常，这可以无缝完成，但有时更新可能会导致你的 pod 被重新安排。因此，我们建议您在作为 Fargate pod 运行应用程序时，创建包含多个副本的部署。

### 定期运行 kube-bench 以验证合规性 Kubernetes 的 CIS 基准测试

kube-bench 是 Aqua 推出的一个开源项目，它根据 Kubernetes 的 CIS 基准测试来评估你的集群。该基准测试描述了保护非托管 Kubernetes 集群的最佳实践。CIS Kubernetes 基准测试包括控制平面和数据平面。由于 Amazon EKS 提供了完全托管的控制平面，因此并非所有来自 CIS Kubernetes 基准测试的建议都适用。为了确保该范围反映亚马逊 EKS 的实施方式，AWS 创建了 CIS 亚马逊 EKS 基准测试。EKS 基准测试继承了 CIS Kubernetes 基准测试，并提供了来自社区的其他意见，其中包括 EKS 集群的具体配置注意事项。

对 EKS 集群运行 kube-bench 时，请按照 Aqua Security 中的这些说明进行操作。欲了解更多信息，请参阅 CIS Amazon EKS 基准测试简介。

### 尽量减少对工作节点的访问

当您需要远程访问主机时，请使用 SSM 会话管理器，而不是启用 SSM 会话管理器。与可能丢失、复制或共享的 SSH 密钥不同，会话管理器允许您使用 IAM 控制对 EC2 实例的访问权限。此外，它还提供在实例上运行的命令的审计跟踪和日志。

自 2020 年 8 月 19 日起，托管节点组支持自定义 AMI 和 EC2 启动模板。这允许您将 SSM 代理嵌入 AMI 或在工作节点被引导时安装它。如果您不想修改优化的 AMI 或 ASG 的启动模板，则可以 DaemonSet 按照本示例中的方式安装 SSM 代理。

基于 SSM 的 SSH 访问的最低 IAM 策略

AmazonSSMManagedInstanceCoreAWS 托管策略包含许多权限， RunCommand 如果您只是想避开 SSH 访问，SSM 会话管理器/SSM 就不需要这些权限。特别值得关注的是允许角色访问参数存储中的所有参数（包括 SecureStrings 配置了 AWS 托管 KMS 密钥的情况下）的*权限。ssm:GetParameter(s)

以下 IAM 策略包含允许通过 SSM Systems Manager 访问节点的最低权限集。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnableAccessViaSSMSessionManager",
      "Effect": "Allow",
      "Action": [
        "ssmmessages:OpenDataChannel",
        "ssmmessages:OpenControlChannel",
        "ssmmessages:CreateDataChannel",
        "ssmmessages:CreateControlChannel",
        "ssm:UpdateInstanceInformation"
      ],
      "Resource": "*"
    },
    {
      "Sid": "EnableSSMRunCommand",
      "Effect": "Allow",
      "Action": [
        "ssm:UpdateInstanceInformation",
        "ec2messages:SendReply",
        "ec2messages:GetMessages",
        "ec2messages:GetEndpoint",
        "ec2messages:FailMessage",
        "ec2messages:DeleteMessage",
        "ec2messages:AcknowledgeMessage"
      ],
      "Resource": "*"
    }
  ]
}
```

有了这个策略并安装了会话管理器插件，你就可以运行了

```
aws ssm start-session --target [INSTANCE_ID_OF_EKS_NODE]
```

访问该节点。

> **Note**
>
> 您可能还需要考虑添加权限以启用会话管理器日志记录。

### 将工作人员部署到私有子网

通过将工作人员部署到私有子网，可以最大限度地减少他们接触经常发起攻击的 Internet 的机会。从 2020 年 4 月 22 日起，托管节点组中节点的公有 IP 地址的分配将由其部署到的子网控制。在此之前，托管节点组中的节点会被自动分配一个公有 IP。如果您选择将工作节点部署到公有子网，请实施限制性的 AWS 安全组规则以限制其暴露风险。

### 运行 Amazon Inspector 来评估主机是否存在漏洞、漏洞以及与最佳实践的偏差

您可以使用 Amazon Inspec tor 检查您的节点是否存在意外网络访问以及底层 Amazon EC2 实例上是否存在漏洞。

只有安装并启用了亚马逊 EC2 Systems Manager (SSM) 代理，Amazon Inspector 才能为您的亚马逊 EC2 实例提供常见漏洞和风险敞口 (CVE) 数据。该代理已预安装在多个亚马逊系统映像 (AMI) 上，包括 EKS 优化的亚马逊 Linux AMI。不管 SSM 代理状态如何，都会对您的所有 Amazon EC2 实例进行网络可访问性问题扫描。有关为 Amazon EC2 配置扫描的更多信息，请参阅扫描 Amazon EC2 实例。

> **Important**
>
> Inspector 无法在用于运行 Fargate 容器的基础设施上运行。

## 替代方案

### 运行 SELinux

> **Note**
>
> 可在红帽企业 Linux (RHEL)、CentOS、Bottlerocket 和亚马逊 Linux 2023 上线

SELinux 提供了额外的安全层，使容器彼此隔离，并与主机隔离。SELinux 允许管理员对每个用户、应用程序、进程和文件实施强制访问控制 (MAC)。可以将其视为一种支持，它根据一组标签将可以针对特定资源执行的操作限制在特定资源上。在 EKS 上，SELinux 可用于防止容器访问彼此的资源。

容器 SELinux 策略是在容器 selinux 软件包中定义的。Docker CE 需要此软件包（及其依赖项），这样 Docker（或其他容器运行时）创建的进程和文件才能在有限的系统访问权限下运行。容器利用作为别名的`container_t`标签`svirt_lxc_net_t`。这些策略可以有效地防止容器访问主机的某些功能。

当你为 Docker 配置 SELinux 时，Docker 会自动将工作负载标记`container_t`为一种类型，并为每个容器提供唯一的 MCS 级别。这将使容器相互隔离。如果您需要更宽松的限制，可以在 SELinux 中创建自己的配置文件，为容器授予文件系统特定区域的权限。这与 PSP 类似，因为您可以为不同的配置文件创建不同的配置文件。 containers/pods例如，您可以为具有一组限制性控制的常规工作负载设置一个配置文件，为需要特权访问权限的内容设置另一个配置文件。

容器版 SELinux 有一组可以配置为修改默认限制的选项。您可以根据需要启用或禁用以下 SELinux 布尔值：

| 布尔值 | 默认值 | 说明 |
| --- | --- | --- |
| `container_connect_any` | `off` | 允许容器访问主机上的特权端口。例如，如果您有一个容器需要将端口映射到主机上的 443 或 80。 |
| `container_manage_cgroup` | `off` | 允许容器管理 cgroup 配置。例如，运行 systemd 的容器需要将其启用。 |
| `container_use_cephfs` | `off` | 允许容器使用 ceph 文件系统。 |

默认情况下，允许 read/execute 容器从中读取大部分内容`/etc`。`/usr`下方`/var/lib/docker`和带有标签`/var/lib/containers`的文件`container_var_lib_t`。要查看默认标签的完整列表，请参阅 con tainer.fc 文件。

```
docker container run -it \
  -v /var/lib/docker/image/overlay2/repositories.json:/host/repositories.json \
  centos:7 cat /host/repositories.json
```

```
# cat: /host/repositories.json: Permission denied

docker container run -it \
  -v /etc/passwd:/host/etc/passwd \
  centos:7 cat /host/etc/passwd
# cat: /host/etc/passwd: Permission denied
```

标有标签的文件container_file_t是唯一可由容器写入的文件。如果要使卷装入可写入，则需要:Z在末尾指定:z或。

- :z将重新标记文件以便容器可以 read/write
- :Z将重新标记文件以便只有容器可以 read/write

```
ls -Z /var/lib/misc
# -rw-r--r--. root root system_u:object_r:var_lib_t:s0   postfix.aliasesdb-stamp

docker container run -it \
  -v /var/lib/misc:/host/var/lib/misc:z \
  centos:7 echo "Relabeled!"

ls -Z /var/lib/misc
#-rw-r--r--. root root system_u:object_r:container_file_t:s0 postfix.aliasesdb-stamp
```

```
docker container run -it \
  -v /var/log:/host/var/log:Z \
  fluentbit:latest
```

在 Kubernetes 中，重新标记略有不同。您可以指定自定义 MCS 标签来运行 pod，而不是让 Docker 自动重新标记文件。支持重新标记的卷将自动重新标记，以便可以访问。带有匹配的 MCS 标签的 Pod 将能够访问该卷。如果您需要严格隔离，请为每个 pod 设置不同的 MCS 标签。

```
securityContext:
  seLinuxOptions:
    # Provide a unique MCS label per container
    # You can specify user, role, and type also
    # enforcement based on type and level (svert)
    level: s0:c144:c154
```

在此示例中，s0:c144:c154对应于分配给允许容器访问的文件的 MCS 标签。

在 EKS 上，您可以创建允许特权容器运行的策略，例如 FluentD，并创建 SELinux 策略以允许它在无需重新标记主机目录的情况下从主机上的 var/log /中读取。具有相同标签的 Pod 将能够访问相同的主机卷。

我们已经为亚马逊 EKS 实施了示例 AMI，这些示例在 CentOS 7 和 RHEL 7 上配置了 SELinux。开发这些 AMI 是为了演示符合高度监管客户要求的示例实施。

> **Warning**
>
> SELinux 将忽略类型不受限制的容器。

## 工具和资源

- SELinux Kubernetes RBAC 和应用程序发布安全政策 On-prem
- Kubernetes 的迭代强化
- 审计 2Allow
- SeAlert
- 使用 Udica 为容器生成 SELinux 策略描述了一种工具，该工具可以查看 Linux 功能、端口和挂载点的容器规格文件，并生成一组允许容器正常运行的 SELinux 规则
- AMI 强化操作手册，用于强化操作系统以满足不同的监管要求
- Keiko Upgrade Manager 是一个来自 Intuit 的开源项目，用于协调工作节点的轮换。
- Sysdig 安全
- eksctl

# 合规

合规是 AWS 及其服务使用者的共同责任。一般而言，AWS负责 “云安全”，而其用户负责 “云安全”。界定 AWS 及其用户应负责的界限将因服务而异。例如，在 Fargate 中，AWS 负责管理其数据中心、硬件、虚拟基础设施 (Amazon EC2) 和容器运行时 (Docker) 的物理安全。Fargate 的用户负责保护容器镜像及其应用程序。在运行必须遵守合规性标准的工作负载时，知道谁应对什么负责，这是一项重要的考虑因素。

下表显示了不同容器服务所遵循的合规计划。

| 合规计划 | 亚马逊 ECS Orchestrator | 亚马逊 EKS 管弦乐器 | ECS Fargate | Amazon ECR |
|---|---|---|---|---|
| PCI DSS 1 级 | 1 | 1 | 1 | 1 |
| 符合 HIPAA 要求 | 1 | 1 | 1 | 1 |
| SOC I | 1 | 1 | 1 | 1 |
| SOC II | 1 | 1 | 1 | 1 |
| SOC II | 1 | 1 | 1 | 1 |
| ISO 27001:2013 | 1 | 1 | 1 | 1 |
| ISO 9001:2015 | 1 | 1 | 1 | 1 |
| ISO 27017:2015 | 1 | 1 | 1 | 1 |
| ISO 27018:2019 | 1 | 1 | 1 | 1 |
| IRAP | 1 | 1 | 1 | 1 |
| FedRamp 中等 () East/ West | 1 | 1 | 0 | 1 |
| FedRamp High () GovCloud | 1 | 1 | 0 | 1 |
| DOD CC SRG | 1 | DISA 评论 (IL5) | 0 | 1 |
| HIPAA BAA | 1 | 1 | 1 | 1 |
| MTCS | 1 | 1 | 0 | 1 |
| C5 | 1 | 1 | 0 | 1 |
| K-ISMS | 1 | 1 | 0 | 1 |
| ENS High | 1 | 1 | 0 | 1 |

| 合规计划 | 亚马逊 ECS Orchestrator | 亚马逊 EKS 管弦乐器 | ECS Fargate | Amazon ECR |
|---|---|---|---|---|
| OSPAR | 1 | 1 | 0 | 1 |
| HITRUST CSF | 1 | 1 | 1 | 1 |

合规性状态会随着时间的推移而变化。要了解最新状态，请务必参阅 https://aws.amazon.com/compliance/services-in-scope/.

有关云认证模式和最佳实践的更多信息，请参阅 AWS 白皮书《安全采用云的认证模型》

## 向左移动

左移的概念涉及在软件开发生命周期的早期发现违反政策的行为和错误。从安全角度来看，这可能非常有益。例如，开发人员可以在将应用程序部署到集群之前修复其配置问题。尽早发现这样的错误将有助于防止部署违反策略的配置。

### 政策即代码

政策可以看作是一套管理行为的规则，即允许的行为或禁止的行为。例如，你可能有一个政策，规定所有 Dockerfile 都应包含一个 USER 指令，该指令使容器以非 root 用户身份运行。作为一份文件，这样的政策可能很难发现和执行。随着需求的变化，它也可能过时。借助策略即代码 (PaC) 解决方案，您可以自动执行安全、合规和隐私控制，以检测、预防、减少和抵消已知和持续存在的威胁。此外，它们还为您提供了编纂策略并像管理其他代码工件一样管理策略的机制。这种方法的好处是，你可以重复使用你的 DevOps 和 GitOps策略，在各个 Kubernetes 集群中管理和一致地应用策略。有关 PaC 选项和 PSP 未来的信息，请参阅 P od Secur ity。

### 在部署之前，使用管道中的策略即代码工具检测违规行为

- OPA 是一个开源政策引擎，是 CNCF 的一部分。它用于制定政策决策，可以以各种不同的方式运行，例如作为语言库或服务。OPA 策略是用一种名为 Rego 的域特定语言 (DSL) 编写的。虽然 OPA 通常作为 Kubernetes 动态准入控制器的一部分作为 Gatekeeper 项目运行，但它也可以整合到你的管道中。 CI/CD 这使开发人员能够在发布周期的早期获得有关其配置的反馈，从而帮助他们在投入生产之前解决问题。可以在该项目的 GitHub存储库中找到常用 OPA 策略的集合。
- C@@ onftest 建立在 OPA 之上，它为测试 Kubernetes 配置提供了以开发人员为中心的体验。
- Kyverno 是一款专为 Kubernetes 设计的策略引擎。在 Kyverno 中，策略作为 Kubernetes 资源进行管理，无需使用任何新语言即可编写策略。这允许使用熟悉的工具，例如 kubectl、git 和 kustomize

来管理策略。Kyverno 策略可以验证、变异和生成 Kubernetes 资源，并确保 OCI 映像供应链的安全。Kyverno CLI 可用于测试策略和验证作为管道一 CI/CD 部分的资源。所有Kyverno社区政策都可以在K yverno网站上找到，有关使用Kyverno CLI在管道中编写测试的示例，请参阅策略存储库。

### 工具和资源

- Amazon EKS 安全沉浸式研讨会——合规性
- kube-bench
- docker-bench 安全
- AWS Inscetor
- Kubernetes 安全评论 Kubernetes 的第三方安全评估 1.13.4 (2019)
- NeuVector 由 SUSE 开源提供的零信任容器安全平台提供合规性报告和自定义合规性检查

## 事件响应和取证

您对事件做出快速反应的能力有助于最大限度地减少漏洞造成的损失。拥有一个可靠的警报系统来警告你注意可疑行为是制定良好的事件响应计划的第一步。当确实发生事故时，你必须迅速决定是销毁和更换受影响的集装箱，还是隔离和检查集装箱。如果您选择隔离集装箱作为取证调查和根本原因分析的一部分，则应遵循以下一系列活动：

### 事件响应计划示例

#### 识别违规的 Pod 和工作节点

你的第一步应该是隔离损坏。首先确定漏洞发生的位置，然后将该 Pod 及其节点与基础设施的其余部分隔离开来。

#### 使用工作负载名称识别违规的 Pod 和工作节点

如果您知道违规 Pod 的名称和命名空间，则可以按如下方式识别运行该 Pod 的工作节点：

```
kubectl get pods <name> --namespace <namespace> -o=jsonpath='{.spec.nodeName}{"\n"}'
```

如果工作负载资源（例如 Deployment）遭到破坏，则属于该工作负载资源的所有 pod 都可能遭到破坏。使用以下命令列出工作负载资源的所有 pod 以及它们正在运行的节点：

```
selector=$(kubectl get deployments <name> \
 --namespace <namespace> -o json | jq -j \
'.spec.selector.matchLabels | to_entries | .[] | "\(.key)=\(.value)"')

kubectl get pods --namespace <namespace> --selector=$selector \
-o json | jq -r '.items[] | "\(.metadata.name) \(.spec.nodeName)"'
```

上面的命令用于部署。您可以对其他工作负载资源（例如副本集、statefulsets 等）运行相同的命令。

## 使用服务账号名称识别违规的 Pod 和工作节点

在某些情况下，您可能会发现服务帐号遭到入侵。使用已识别服务帐号的 pod 很可能遭到入侵。您可以使用以下命令识别使用服务帐户的所有 Pod 以及它们正在运行的节点：

```
kubectl get pods -o json --namespace <namespace> | \
    jq -r '.items[] |
    select(.spec.serviceAccount == "<service account name>") |
    "\(.metadata.name) \(.spec.nodeName)"'
```

## 识别包含易受攻击或受损图像的 Pod 以及工作节点

在某些情况下，您可能会发现集群上的 pod 中使用的容器镜像是恶意的或已被入侵的。如果容器镜像被发现包含恶意软件、是已知的错误镜像或有已被利用的 CVE，则该容器镜像为恶意镜像或已被攻击。你应该考虑所有使用容器镜像的 pod 都已受到损害。你可以使用以下命令使用镜像来识别 Pod 以及它们正在运行的节点：

```
IMAGE=<Name of the malicious/compromised image>

kubectl get pods -o json --all-namespaces | \
    jq -r --arg image "$IMAGE" '.items[] |
    select(.spec.containers[] | .image == $image) |
    "\(.metadata.name) \(.metadata.namespace) \(.spec.nodeName)"'
```

## 通过创建网络策略来隔离 Pod 的所有入口和出口流量

拒绝所有流量规则可以切断与 Pod 的所有连接，从而帮助阻止已经在进行的攻击。以下网络策略将适用于带有标签的 pod app=web。

```
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  - Egress
```

**Important**

如果攻击者获得了底层主机的访问权限，则网络策略可能被证明是无效的。如果您怀疑发生了这种情况，则可以使用 AWS 安全组将受感染的主机与其他主机隔离开来。更改主机的安全组时，请注意，这将影响该主机上运行的所有容器。

### 如有必要，撤消分配给 pod 或工作节点的临时安全证书

如果已为工作节点分配了允许 Pod 访问其他 AWS 资源的 IAM 角色，请将这些角色从实例中移除，以防止攻击造成进一步损害。同样，如果已为容器组分配了 IAM 角色，请评估您是否可以在不影响其他工作负载的情况下，从该角色安全删除 IAM 策略。

### 封锁工作节点

通过封锁受影响的工作节点，你就是在通知调度器避免将 pod 调度到受影响的节点上。这将允许您在不中断其他工作负载的情况下移除节点进行取证研究。

**Note**

本指南不适用于 Fargate，其中每个 Fargate 吊舱都在自己的沙盒环境中运行。与其封锁，不如应用拒绝所有入口和出口流量的网络策略来封存受影响的 Fargate 吊舱。

### 在受影响的工作节点上启用终止保护

攻击者可能试图通过终止受影响的节点来抹去他们的不当行为。启用终止保护可以防止这种情况发生。实例缩减保护将保护节点免受缩容事件的影响。

> **Warning**
>
> 您无法在竞价型实例上启用终止保护。

### 在违规行为 Pod/Node 上贴上标签，表明这是正在进行的调查的一部分

这将警告集群管理员，在调查完成 Pods/Nodes 之前不要篡改受影响的人。

### 捕获工作节点上的易失性工件

- 捕获操作系统内存。这将捕获 Docker 守护程序（或其他容器运行时）及其每个容器的子进程。这可以使用 Lim E 和 Volatility 等工具来实现，也可以通过更高级的工具来实现，例如基于这些工具构建的 Amazon EC2 自动取证协调器。
- 对@@ 正在运行的进程和打开的端口执行 netstat 树转储。这将捕获每个容器的 docker 守护程序及其子进程。
- 在@@ 证据被更改之前，运行命令以保存容器级别的状态。您可以使用容器运行时的功能来捕获有关当前正在运行的容器的信息。例如，使用 Containerd，您可以执行以下操作：
  - `crictl ps`用于正在运行的进程。
  - `crictl logs CONTAINER`用于守护程序级别保存的日志。

  使用 nerdctl CLI 代替（例如），使用 cont ainerd 也可以实现同样`docker`的效果。`nerdctl inspect`根据容器运行时的不同，还有一些其他命令可用。例如，Docker `docker diff` 必须查看容器文件系统的更改或`docker checkpoint`保存包括易失性内存 (RAM) 在内的所有容器状态。有关与 contain erd 或运行时类似功能的讨论，请参阅这篇 Kubernetes 博客文章。 CRI-O
- 暂停容器以进行取证捕获。
- 为实例的 EBS 卷创建快照。

### 重新部署受感染的 Pod 或工作负载资源

收集数据进行取证分析后，您可以重新部署受感染的 pod 或工作负载资源。

首先推出针对已被破坏的漏洞的修复程序，然后启动新的替换 pod。然后删除有漏洞的 pod。

如果易受攻击的 Pod 由更高级别的 Kubernetes 工作负载资源（例如 Deployment 或 DaemonSet）管理，则删除它们将安排新的 Pod。因此，易受攻击的吊舱将再次启动。在这种情况下，您应该在修复漏洞后部署新的替代工作负载资源。然后，您应该删除有漏洞的工作负载。

## 建议

### 查看 AWS 安全事件响应白皮书

虽然本节简要概述了处理可疑安全漏洞的一些建议，但白皮书《AWS 安全事件响应》对这一主题进行了详尽的介绍。

### 练习安全游戏日

将您的安全从业人员分为 2 个小组：红色和蓝色。红队将专注于探测不同的系统是否存在漏洞，而蓝队将负责防御这些漏洞。如果你没有足够的安全从业人员来创建单独的团队，可以考虑雇用一个了解 Kubernetes 漏洞利用的外部实体。

Kubesploit 是一个渗透测试框架 CyberArk ，你可以用它来进行游戏日。与其他扫描集群漏洞的工具不同，kubesploit 模拟的是真实世界的攻击。这让你的蓝队有机会练习对攻击的反应并衡量其有效性。

### 对你的集群运行渗透测试

定期攻击自己的集群可以帮助您发现漏洞和错误配置。在开始之前，请先遵循渗透测试指南，然后再对集群进行测试。

## 工具和资源

- kube-hunter，一款适用于 Kubernetes 的渗透测试工具。
- Gremlin，一款混沌工程工具包，可用于模拟针对应用程序和基础设施的攻击。
- 攻击和防御 Kubernetes 设施
- kubesploit
- NeuVector 由 SUSE 开源提供的零信任容器安全平台提供漏洞和风险报告以及安全事件通知
- 高级持续威胁
- Kubernetes 实用攻击和防御
- 利用 RBAC 权限破坏 Kubernetes 集群

# 映像安全

您应该将容器映像视为抵御“攻击”的第一道防线。不安全、构造不佳的映像可能允许攻击者逃离容器的边界并获得对主机的访问权限。一旦进入主机，攻击者就可以访问敏感信息，或者在集群内或使用您的 AWS 账户横向移动。以下最佳实践将有助于降低发生这种情况的风险。

# 建议

## 创建最少的图像

首先从容器映像中移除所有多余的二进制文件。如果你使用的是来自 Dockerhub 的不熟悉镜像，请使用像 D iv e 这样的应用程序检查镜像，它可以向你显示容器每个层的内容。删除所有带有 SETUID 和 SETGID 位的二进制文件，因为它们可以用来升级权限，并考虑删除所有可用于邪恶目的的 shell 和实用程序，例如 nc 和 curl。您可以使用以下命令查找带有 SETUID 和 SETGID 位的文件：

```
find / -perm /6000 -type f -exec ls -ld {} \;
```

要从这些文件中删除特殊权限，请在您的容器映像中添加以下指令：

```
RUN find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true
```

通俗地说，这被称为去除你的形象。

## 使用多阶段构建

使用多阶段构建是一种创建最少图像的方法。通常，多阶段构建用于自动化持续集成周期的某些部分。例如，多阶段构建可用于整理源代码或执行静态代码分析。这使开发人员有机会获得近乎即时的反馈，而不必等待管道执行。 Multi-stage 从安全角度来看，构建具有吸引力，因为它们允许您最大限度地减少推送到容器注册表的最终映像的大小。缺乏构建工具和其他无关二进制文件的容器映像通过减少映像的攻击面来改善您的安全状况。有关多阶段构建的更多信息，请参阅 Docker 的多阶段构建文档。

## 为容器映像创建软件物料清单 (SBOM)

“软件物料清单” (SBOM) 是构成容器映像的软件工件的嵌套清单。SBOM 是软件安全和软件供应链风险管理的关键组成部分。生成 SBOM 并将其存储在中央存储库中并扫描 SBOM 是否存在漏洞有助于解决以下问题：

- 可见性：了解哪些组件构成了您的容器镜像。存储在中央存储库中，可以随时对SBOM进行审计和扫描，甚至可以在部署后进行审核和扫描，以检测和响应新漏洞，例如未修补的漏洞。
- 来源验证：确保对工件的来源和来源的现有假设是正确的，并且在构建或交付过程中工件或其随附的元数据没有被篡改。
- 可信度：保证可以信任给定的工件及其内容来做它声称要做的事情，即适合于某一目的。这包括判断代码是否可以安全执行，并就与执行代码相关的风险做出明智的决定。通过创建经过验证的管道执行

报告以及经过验证的 SBOM 和经认证的 CVE 扫描报告来确保可信度，从而向图像的消费者保证，该图像实际上是通过带有安全组件的安全手段（管道）创建的。

- 依赖信任验证：递归检查工件的依赖树，以确定其使用的工件的可信度和来源。SBOM 中的漂移可以帮助检测恶意活动，包括未经授权、不受信任的依赖关系、渗透尝试。

以下工具可用于生成 SBOM：

- Amazon Inspector 可用于创建和导出 SBOM。
- Anchore 的 Syft 也可以用于生成 SBOM。为了更快地扫描漏洞，可以将为容器映像生成的 SBOM 用作扫描的输入。然后，对 SBOM 和扫描报告进行证明，并将其附加到映像上，然后将映像推送到中央 OCI 存储库（例如 Amazon ECR）进行审查和审计。

查看 CNCF 软件供应链最佳实践指南，了解有关保护软件供应链的更多信息。

### 定期扫描图像以查找漏洞

与虚拟机镜像一样，容器镜像可能包含有漏洞的二进制文件和应用程序库，或者随着时间的推移而出现漏洞。防范漏洞的最佳方法是定期使用图像扫描仪扫描图像。存储在 Amazon ECR 中的图像可以通过推送或按需扫描（在 24 小时内扫描一次）。ECR 目前支持两种扫描类型：基本扫描和增强扫描。基本扫描免费利用 Clair 开源图像扫描解决方案。增强型扫描使用 Amazon Inspector 提供自动连续扫描，但需要额外付费。扫描图像后，结果将记录到事件流中，以便在 ECR 中执行。EventBridge您还可以从 ECR 控制台中查看扫描结果。应删除或重建具有 “高” 或 “严重” 漏洞的图像。如果已部署的映像出现漏洞，则应尽快将其更换。

了解存在漏洞的映像的部署位置对于确保环境安全至关重要。虽然你可以自己构建图像跟踪解决方案，但已经有几种商业产品提供了开箱即用的此功能和其他高级功能，包括：

- Grype
- 帕洛阿尔托——Prisma Cloud (twistcli)
- 水族
- Kubei
- Trivy
- Snyk

Kubernetes 验证 webhook 也可以用来验证镜像是否没有严重漏洞。验证网络挂钩在 Kubernetes API 之前被调用。它们通常用于拒绝不符合 webhook 中定义的验证标准的请求。这是一个无服务器

Webhook 的示例，它调用 ECR describe ImageScanFindings API 来确定 pod 是否正在提取存在严重漏洞的图像。如果发现漏洞，Pod 将被拒绝，并以事件形式返回一条包含 CVE 列表的消息。

## 使用认证来验证工件的完整性

认证是一种加密签名的“声明”，它声称某件事——一个“谓词”，例如管道运行、SBOM 或漏洞扫描报告对另一件事是正确的——“主题”，即容器镜像。

认证可以帮助用户验证工件是否来自软件供应链中的可信来源。例如，我们可能在不知道镜像中包含的所有软件组件或依赖项的情况下使用容器镜像。但是，如果我们相信容器镜像的制作者所说的关于存在什么软件，我们就可以使用制作人的认证来依赖该工件。这意味着我们可以继续在工作流程中安全地使用工件，而不必自己进行分析。

- 可以使用 AW S Signer 或 Sigstore 共同签名来创建证明。
- Kubernetes 准入控制器（例如 Kyverno）可用于验证认证。
- 请参阅此研讨会，详细了解 AWS 上使用开源工具进行软件供应链管理的最佳实践，主题包括创建和附加对容器映像的认证。

## 为 ECR 存储库创建 IAM 策略

如今，一个组织在共享的 AWS 账户中拥有多个开发团队独立运行的情况并不少见。如果这些团队不需要共享资产，则可能需要创建一组 IAM 策略来限制对每个团队可以与之交互的存储库的访问权限。实现这一点的一个好方法是使用 ECR 命名空间。命名空间是一种将相似的存储库组合在一起的方法。例如，团队 A 的所有注册表都可以以 team-a/ 开头，而团队 B 的注册表可以使用 team-b/ 前缀。限制访问的策略可能如下所示：

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowPushPull",
      "Effect": "Allow",
      "Action": [
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability",
        "ecr:PutImage",
        "ecr:InitiateLayerUpload",
        "ecr:UploadLayerPart",
```

```
        "ecr:CompleteLayerUpload"
      ],
      "Resource": [
        "arn:aws:ecr:us-east-1:123456789012:repository/team-a/*"
      ]
    }
  ]
}
```

## 考虑使用 ECR 私有端点

ECR API 有一个公共端点。因此，只要请求已通过 IAM 的身份验证和授权，就可以从互联网访问 ECR 注册表。对于那些需要在集群 VPC 缺少 Internet Gateway (IGW) 的沙盒环境中操作的用户，您可以为 ECR 配置私有终端节点。通过创建私有终端节点，您可以通过私有 IP 地址私密访问 ECR API，而不是通过互联网路由流量。有关此主题的更多信息，请参阅 Amazon ECR 接口 VPC 终端节点。

## 为 ECR 实施端点策略

的默认终端节点策略允许访问一个区域内的所有 ECR 存储库。这可能允许通过 attacker/insider 将数据打包为容器镜像并将其推送到另一个 AWS 账户的注册表来泄露数据。要降低这种风险，需要创建一个端点策略，限制 API 对 ECR 存储库的访问。例如，以下策略允许您账户中的所有 AWS 原则对您的 ECR 存储库执行所有操作，且仅允许您的 ECR 存储库执行所有操作：

```
{
  "Statement": [
    {
      "Sid": "LimitECRAccess",
      "Principal": "*",
      "Action": "*",
      "Effect": "Allow",
      "Resource": "arn:aws:ecr:<region>:<account_id>:repository/*"
    }
  ]
}
```

您可以通过设置一个使用新`PrincipalOrgID`属性的条件来进一步增强这一点，该条件将根据不属于您的 AWS 组织的 IAM 原则阻止 pushing/pulling 映像。有关更多详细信息，请参阅 aws: PrincipalOrgID。建议对 `com.amazonaws.<region>.ecr.dkr` 和 `com.amazonaws.<region>.ecr.api` 端点应用相同的策略。由于 EKS 从 ECR 中提取 kube-proxy、coredns 和 aws-node 的图像，因此您需要添加注册表的账户 ID，例如`602401143452.dkr.ecr.us-west-2.amazonaws.com/` 添加到端

点策略中的资源列表中，或者更改策略以允许从您的账户 ID 提取和限制推送到您的账户 ID。下表显示了出售 EKS 映像的 AWS 账户与集群区域之间的对应关系。

| 账号 | Region |
| --- | --- |
| 602401143452 | 除下面列出的区域以外的所有商业区域 |
| — | — |
| 800184023465 | ap-east-1-亚太地区（香港） |
| 558608220178 | me-south-1-中东（巴林） |
| 918309763551 | cn-north-1-中国（北京） |
| 961992271922 | cn-northwest-1-中国（宁夏） |

有关使用终端节点策略的更多信息，请参阅使用 VPC 终端节点策略控制 Amazon ECR 访问权限。

## 为 ECR 实施生命周期策略

NIST《应用程序容器安全指南》警告 “注册表中存在陈旧映像” 的风险，并指出，随着时间的推移，应删除带有易受攻击、过时软件包的旧映像，以防止意外部署和泄露。每个 ECR 存储库都可以有一个生命周期策略，用于设置镜像何时过期的规则。A WS 官方文档描述了如何设置测试规则、对其进行评估然后应用这些规则。官方文档中有几个生命周期策略示例，显示了筛选存储库中图像的不同方法：

- 按图片年龄或数量筛选
- 按带标签或未加标签的图像进行筛选
- 按图像标签筛选，可以在多个规则中或单个规则中筛选

？？？ + 警告如果从 ECR 中清除长时间运行的应用程序的映像，则在重新部署或水平扩展应用程序时，可能会导致映像提取错误。使用映像生命周期策略时，请确保您有良好的 CI/CD 实践，以使部署及其引用的映像保持最新，并始终根据您的频率创建 [image] 到期规则 releases/deployments。

## 创建一组精选映像

与其允许开发人员创建自己的映像，不如考虑为组织中的不同应用程序堆栈创建一组经过审查的图像。通过这样做，开发人员可以放弃学习如何编写 Dockerfiles，而专注于编写代码。当更改合并到 Master

时， CI/CD 管道可以自动编译资产，将其存储在工件存储库中，然后将工件复制到相应的镜像中，然后再将其推送到 ECR 之类的 Docker 注册表。至少你应该创建一组基础镜像，开发者可以从中创建自己的 Dockerfiles。理想情况下，你要避免从 Dockerhub 提取镜像，因为 1/ 你并不总是知道镜像中有什么，而且 2/ 在前 1000 张镜像中，大约有五分之一存在漏洞。可以在此处找到这些图像及其漏洞的列表。

### 将 USER 指令添加到你的 Dockerfiles 中，以非 root 用户身份运行

正如 pod 安全部分所述，你应该避免以 root 用户身份运行容器。虽然你可以将其配置为 PodSpec 的一部分，但在 Dockerfiles 中使用该USER指令是个好习惯。该USER指令设置运行时使用的 UID RUNENTRYPOINT，或显示在 USE CMD R 指令之后的指令。

### 整理你的 DockerFiles

Linting 可用于验证您的 Dockerfile 是否遵守了一组预定义的指导方针，例如包含USER指令或要求对所有图像进行标记。 dockerfile_lint 是一个开源项目 RedHat ，它验证了常见的最佳实践，并包含一个规则引擎，你可以用它来构建自己的清理 Dockerfiles 的规则。它可以合并到 CI 管道中，因为使用违反规则的 Dockerfiles 构建将自动失败。

### 从头开始构建镜像

在构建镜像时，减少容器镜像的攻击面应该是主要目标。实现此目的的理想方法是创建最少的镜像，其中不包含可用于利用漏洞的二进制文件。幸运的是，Docker 有一种机制可以用来创建镜像。scratch使用像 Go 这样的语言，你可以创建一个静态链接的二进制文件，并在你的 Dockerfile 中引用它，如下例所示：

```
############################
# STEP 1 build executable binary
############################
FROM golang:alpine AS builder# Install git.
# Git is required for fetching the dependencies.
RUN apk update && apk add --no-cache gitWORKDIR $GOPATH/src/mypackage/myapp/COPY . . #
 Fetch dependencies.
# Using go get.
RUN go get -d -v# Build the binary.
RUN go build -o /go/bin/hello

############################
# STEP 2 build a small image
############################
FROM scratch# Copy our static executable.
```

```
COPY --from=builder /go/bin/hello /go/bin/hello# Run the hello binary.
ENTRYPOINT ["/go/bin/hello"]
```

这将创建一个容器镜像，该镜像仅包含您的应用程序，仅包含其他任何内容，因此非常安全。

### 在 ECR 中使用不可变标签

不可变标签会强制您在每次推送到图像存储库时更新图像标签。这可以阻止攻击者在不更改图像标签的情况下使用恶意版本覆盖图像。此外，它还为您提供了一种轻松且唯一地识别图像的方法。

### 签署您的映像、SBOM、管道运行和漏洞报告

当首次引入 Docker 时，还没有用于验证容器镜像的加密模型。在 v2 中，Docker 在镜像清单中添加了摘要。这允许对图像的配置进行哈希处理，并使用哈希值为图像生成 ID。启用图像签名后，Docker 引擎会验证清单的签名，确保内容来自可信来源且未发生任何篡改。下载每个图层后，引擎都会验证图层的摘要，确保内容与清单中指定的内容相匹配。图像签名实际上允许您通过验证与图像相关的数字签名来创建安全的供应链。

我们可以使用 AWS Signer 或 Sigstore Cosign 来签署容器映像、为 SBOM 创建证明、漏洞扫描报告和管道运行报告。这些证明确保了图像的可信度和完整性，它实际上是由可信管道创建的，没有任何干扰或篡改，并且它仅包含经过图像发布者验证和信任的记录在案（在 SBOM 中）的软件组件。这些证明可以附加到容器镜像并推送到存储库。

在下一节中，我们将看到如何使用经过认证的工件进行审计和招生控制员验证。

### 使用 Kubernetes 准入控制器进行图像完整性验证

在使用动态准入控制器将映像部署到目标 Kubernetes 集群之前，我们可以自动验证图像签名和已验证的工件，并且只有在工件的安全元数据符合准入控制器策略时才允许部署。

例如，我们可以编写一个策略，以加密方式验证图像的签名、经过验证的 SBOM、经认证的管道运行报告或经过验证的 CVE 扫描报告。我们可以在策略中写入条件来检查报告中的数据，例如，CVE 扫描不应有任何关键 CVE。只有满足这些条件的映像才允许部署，所有其他部署都将被招生控制器拒绝。

准入控制器的示例包括：

- Kyverno
- OPA 看门人
- Portieris

- 批准
- Kritis
- Grafeas 教程
- 代金券

### 更新容器镜像中的软件包

你应该在 Dockerfiles `apt-get update && apt-get upgrade` 中加入 RUN 来升级镜像中的软件包。尽管升级需要您以 root 用户身份运行，但这发生在映像构建阶段。该应用程序不需要以 root 用户身份运行。您可以安装更新，然后使用 USER 指令切换到其他用户。如果您的基础映像以非 root 用户身份运行，请切换到 root 用户并返回；不要仅仅依靠基础映像的维护者来安装最新的安全更新。

运行`apt-get clean`以从中删除安装程序文件`/var/cache/apt/archives/`。你也可以在安装软件包`rm -rf /var/lib/apt/lists/*`后运行。这将删除索引文件或可供安装的软件包列表。请注意，每个软件包管理器的这些命令可能有所不同。例如：

```
RUN apt-get update && apt-get install -y \
    curl \
    git \
    libsqlite3-dev \
    && apt-get clean && rm -rf /var/lib/apt/lists/*
```

## 工具和资源

- Amazon EKS 安全沉浸式研讨会-图像安全
- docker-slim 构建安全的最小镜像
- dockle 验证你的 Dockerfile 是否符合创建安全镜像的最佳实践
- dockerfile-lint 基于 Dockerfiles 规则的 lint er
- hadolint 一个聪明的 dockerfile lint er
- Gatekeeper 和 OPA 基于策略的准入控制器
- Kyverno 策略引擎 Kubernetes-native
- in-toto 允许用户验证供应链中的某个步骤是否打算执行，以及该步骤是否由正确的参与者执行
- Notary 一个用于签署容器镜像的项目
- Notary v2
- Grafeas 一个开放的工件元数据 API，用于审计和管理您的软件供应链

- NeuVector 由 SUSE 开源提供的零信任容器安全平台提供容器、镜像和注册表扫描，以查找漏洞、机密和合规性。

# 多账户策略

AWS 建议使用多账户策略和 AWS 组织来帮助隔离和管理您的业务应用程序和数据。使用多账户策略有很多好处：

- 增加了 AWS API 服务配额。配额适用于 AWS 账户，使用多个账户处理工作负载会增加工作负载可用的总配额。
- 更简单的身份和访问管理 (IAM) Access Management 策略。仅向工作负载和支持他们的操作员授予访问自己的 AWS 账户的权限意味着可以减少为实现最小权限原则而制定精细的 IAM 策略所需的时间。
- 改进了 AWS 资源的隔离。根据设计，在一个账户中配置的所有资源在逻辑上都与其他账户中配置的资源隔离。该隔离边界提供了一种限制应用程序相关问题、配置错误或恶意操作风险的方法。如果一个账户中出现问题，可以减轻或消除对其他账户中所含工作负载的影响。
- 更多好处，如 AWS 多账户策略白皮书中所述

以下各节将说明如何使用集中式或去中心化的 EKS 集群方法为 EKS 工作负载实施多账户策略。

## 为多租户集群规划多工作负载账户策略

在多账户 AWS 策略中，属于给定工作负载的资源（例如 S3 存储桶、 ElastiCache 集群和 DynamoDB 表）都是在包含该工作负载的所有资源的 AWS 账户中创建的。这些帐户称为工作负载帐户，EKS 集群部署到称为集群帐户的账户中。下一节将探讨集群账户。将资源部署到专用工作负载账户与将 kubernetes 资源部署到专用命名空间类似。

然后，可以根据软件开发生命周期或其他要求（如果适用）进一步细分工作负载帐户。例如，给定的工作负载可以有一个生产账户、一个开发账户，或者一个用于在特定区域托管该工作负载的实例的账户。更多信息可在本 AWS 白皮书中找到。

在实施 EKS 多账户策略时，您可以采用以下方法：

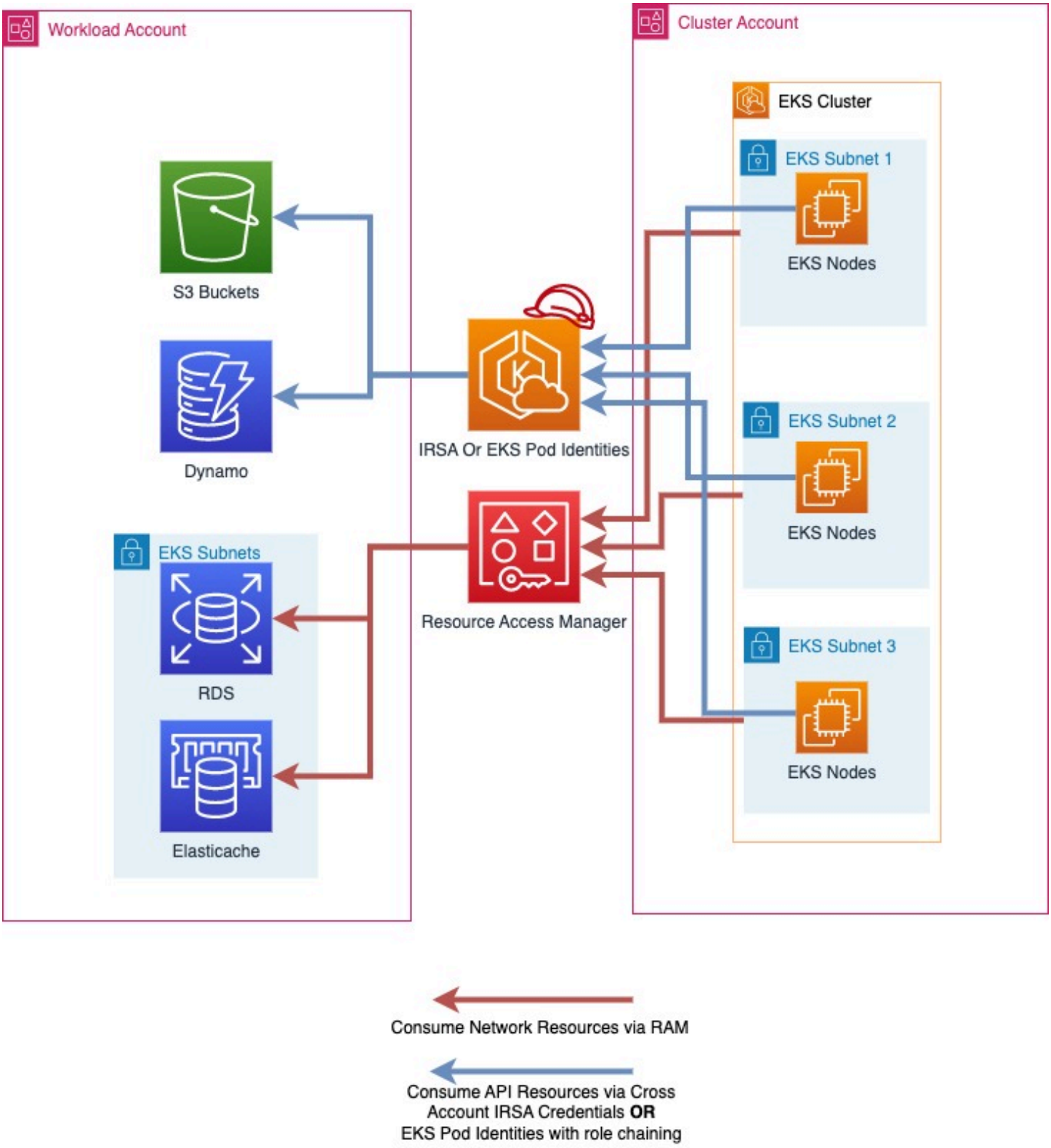
## 集中式 EKS 集群

采用这种方法，您的 EKS 集群将部署在名为的单个 AWS 账户中`Cluster Account`。使用服务账户 (IRSA) 或 EKS Pod 身份的 IAM 角色来提供临时 AWS 证书，使用 AWS Resource Access Manager

(RAM) 来简化网络访问，您可以为多租户 EKS 集群采用多账户策略。集群账户将包含 VPC、子网、EKS 集群、 EC2/Fargate 计算资源（工作节点）以及运行 EKS 集群所需的任何其他网络配置。

在多租户集群的多工作负载账户策略中，AWS 账户通常与 kubernetes 命名空间保持一致，以此作为隔离资源组的机制。在@@ 为多租户 EKS 集群实施多账户策略时，仍应遵循在 EKS 集群内进行租户隔离的最佳实践。

您的 AWS 组织Cluster Accounts中可能有多个，最佳做法是让多个Cluster Accounts符合您的软件开发生命周期需求的多个。对于大规模运行的工作负载，可能需要多个，Cluster Accounts以确保有足够的 kubernetes 和 AWS 服务配额可供所有工作负载使用。




|在上图中，AWS RAM 用于将子网从集群账户共享到工作负载账户。然后，在 EKS 容器中运行的工作负载使用 IRSA 或 EKS Pod 身份和角色链在其工作负载账户中扮演角色并访问其 AWS 资源。

## 为多租户集群实施多工作负载账户策略

使用 AWS Resource Access Manager 共享子网

AWS Resource Access Manager (RAM) 允许您在 AWS 账户之间共享资源。

如果您的 AWS 组织启用了 RAM，则可以将集群账户中的 VPC 子网共享给您的工作负载账户。这将允许将您的工作负载账户拥有的 AWS 资源（例如亚马逊 ElastiCache集群或亚马逊关系数据库服务 (RDS) 数据库）部署到与您的 EKS 集群相同的 VPC 中，并且可供在 EKS 集群上运行的工作负载使用。

要通过 RAM 共享资源，请在集群账户的 AWS 控制台中打开 RAM，然后选择 “资源共享” 和 “创建资源共享”。命名您的资源共享并选择要共享的子网。再次选择 “下一步”，输入要与之共享子网的工作负载帐户的 12 位帐户 ID，再次选择 “下一步”，然后单击 “创建资源共享” 完成操作。完成此步骤后，工作负载帐户可以将资源部署到这些子网中。

也可以通过编程方式创建 RAM 共享，也可以使用基础架构即代码创建。

在 EKS Pod 身份和 IRSA 之间做出选择

在 re: Invent 2023 上，AWS 推出了 EKS Pod Identities，这是一种在 EKS 上向你的 pod 提供临时 AWS 证书的更简单方式。IRSA 和 EKS Pod 身份都是向您的 EKS 容器提供临时 AWS 证书的有效方法，并将继续得到支持。您应该考虑哪种交付方式最能满足您的需求。

在使用一个 EKS 集群和多个 AWS 账户时，IRSA 可以直接在 AWS 账户中扮演角色，而不是直接托管 EKS 集群的账户，而 EKS Pod 身份则需要您配置角色链。有关深入比较，请参阅 E KS 文档。

使用服务账户的 IAM 角色访问 AWS API 资源

服务账户的 IAM 角色 (IRSA) 允许您向在 EKS 上运行的工作负载提供临时的 AWS 证书。IRSA 可用于从集群账户获取工作负载账户中 IAM 角色的临时证书。这允许您在集群账户中的 EKS 集群上运行的工作负载消耗 AWS API 资源，例如工作负载账户中托管的 S3 存储桶，并对 Amazon RDS 数据库或 Amazon EFS 等资源使用 IAM 身份验证。 FileSystems

在工作负载账户中使用 IAM 身份验证的 AWS API 资源和其他资源只能通过同一工作负载账户中的 IAM 角色的证书进行访问，除非支持跨账户访问且已明确启用。

启用 IRSA 进行跨账户访问

要为集群账户中的工作负载启用 IRSA 访问工作负载账户中的资源，您必须先在工作负载账户中创建 IAM OIDC 身份提供商。除了将在工作负载帐户中创建身份提供者之外，还可以使用与设置 IRSA 相同的步骤来完成此操作。

然后，在 EKS 上为工作负载配置 IRSA 时，您可以按照与文档相同的步骤进行操作，但要使用 “示例从其他账户的集群创建身份提供商” 一节中提到的工作负载帐户的 12 位数帐户 ID。

配置完成后，在 EKS 中运行的应用程序将能够直接使用其服务帐户在工作负载帐户中扮演角色，并使用其中的资源。

使用 EKS 容器身份访问 AWS API 资源

EKS Pod 身份提供了另一种向在 EKS 上运行的工作负载提供临时 AWS 证书的方法。EKS Pod Identities 与 EKS 控制平面和集群上代理集成，因此 Pod 无需您创建或管理 IAM OIDC 身份提供商即可接收证书。EKS Pod 身份是支持节点类型上的新工作负载的推荐方法，而 IRSA 仍然是完全支持的替代方案。

使用 EKS Pod 身份，您可以将集群中的 Kubernetes 服务账户与集群所在的 AWS 账户中的 IAM 角色关联起来。EKS 使用此关联来获取该 IAM 角色的临时证书，并将其安全地交付给使用该服务账户的 pod。然后，您的应用程序可以使用标准 AWS 开发工具包和默认凭证链来调用 AWS API；无需自定义凭证提供程序或配置文件。

启用 EKS Pod 身份进行跨账户访问

EKS Pod Identities 通过在工作负载账户中使用目标 IAM 角色和 IAM 角色链原生支持跨账户访问。在为 Kubernetes 服务账户创建 Pod 身份关联时，您可以在集群账户中指定一个 pod IAM 角色，并在工作负载账户中指定目标 IAM 角色。EKS Pod Identity 使用 pod 角色担任目标角色，并将目标角色的临时证书返回给 pod。

有关此方法的详细演练和注意事项，请参阅此 AWS 博客。

用于跨账户访问的 ABAC 和 EKS Pod 身份

作为多账户策略的一部分，使用 EKS Pod Identities 在其他账户中扮演角色（角色链）时，您可以选择为需要访问另一个账户的每个服务账户分配一个唯一的 IAM 角色，或者在多个服务账户中使用通用 IAM 角色并使用 ABAC 来控制其可以访问的账户。

要使用 ABAC 控制哪些服务帐号可以通过角色链接将角色代入另一个账户，您需要创建一个角色信任策略声明，该声明仅允许在存在预期值时由 EKS 集群账户（账户 ID 111122223333）的 pod IAM 角色代入角色。只有在、`eks-cluster-arn`和`kubernetes-namespace`标签都具有预期值的情况下，以下角色信任策略才允许来自 EKS 集群账户的 pod IAM 角色代入角色。`kubernetes-service-account`

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/account-a-role"
            },
            "Action": [
                "sts:AssumeRole",
                "sts:TagSession"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/kubernetes-service-account": "PayrollApplication",
                    "aws:RequestTag/eks-cluster-arn": "arn:aws:eks:us-
east-1:111122223333:cluster/ProductionCluster",
                    "aws:RequestTag/kubernetes-namespace": "PayrollNamespace"
                }
            }
        }
    ]
}
```

您还可以使用 EKS Pod 身份会话策略进一步缩小容器的权限范围，而无需创建其他 IAM 角色。会话策略在 EKS Pod Identity 担任角色时应用，由此产生的权限是角色策略和会话策略的交叉点；有关注意事项和详细步骤，请参阅 AWS Containers 博客 A mazon EKS Pod Identity 的会话策略。

使用此策略时，最佳做法是确保常见 IAM 角色只有`sts:AssumeRole`和`sts:TagSession`权限，没有其他 AWS 访问权限。

使用 ABAC 时，重要的是要控制谁能够将 IAM 角色、用户和 STS 会话标记给那些有严格需求的人。能够设置`kubernetes-`或`eks-`标签的人可能能够设置与 EKS Pod Identity 角色链接期间传递的标签相同的标签，并且可以升级其权限。您可以使用 IAM 策略或服务控制策略 (SCP) 限制谁有权设置这些标签；例如控件，请参阅ProtectPodIdentitiesTagsOnRolesAndUsers和STS-Protect-EKS-pod-identities-tags。

在 EKS Pod 身份和 IRSA 之间做出选择

IRSA 和 EKS Pod 身份都是向您的 EKS 工作负载提供临时 AWS 证书的有效选项。对于在支持的节点类型上运行的新应用程序，建议使用 EKS Pod 身份，而 IRSA 非常适合您已经安装了 OIDC 和 IRSA 或者在 EKS Pod Identitions 不支持的平台上运行的情况。

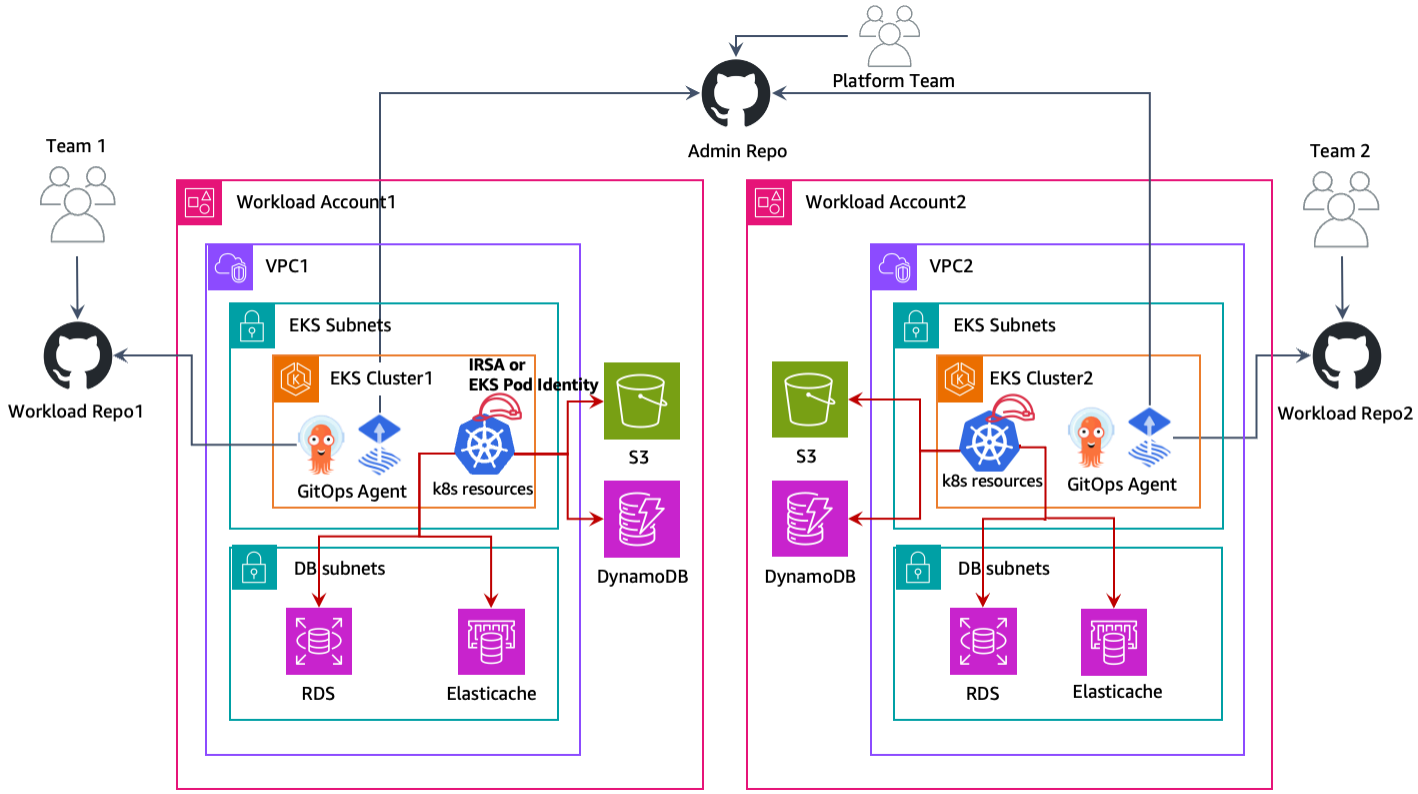
在决定要使用哪种方式时，请注意以下因素：

- 在以下情况下选择 EKS Pod 身份：
  - 您正在设计新的工作负载，希望避免创建和管理 IAM OIDC 身份提供商。
  - 您希望使用目标 IAM 角色为跨账户访问提供原生支持，而无需向 pod 添加自定义凭证脚本或 AWS 配置文件。
  - 您希望集群管理员管理哪些 Kubernetes 服务账户可以扮演哪些角色，而 IAM 管理员则管理这些角色的权限。
  - 您希望凭证自动售卖能够高效扩展，避免达到 IAM 配额限制。
- 在以下情况下选择 IRSA：
  - 您已经成功使用了 IRSA，并且有 OIDC 提供商和 IAM 角色的标准模式。
  - 您的工作负载在不支持 EKS 容器身份的环境中运行，例如 AWS Fargate、Windows 节点或使用不受支持的 AWS 软件开发工具包的应用程序。
  - 您需要为工作负载帐户中的角色建立直接 OIDC-based 联合模型，并且您的安全控制已经围绕 OIDC 提供商构建。

IRSA 和 EKS Pod 身份都支持多账户策略。你可以在集群中一致使用这两种方法，也可以采用混合模式，即传统工作负载继续使用 IRSA，而新的工作负载使用 EKS Pod Identities。

## De-centralized EKS 集群

在这种方法中，EKS 集群部署到各自的工作负载 AWS 账户，并与其他 AWS 资源（例如 Amazon S3 存储桶、VPC、Amazon DynamoDB 表等）并存，每个工作负载账户都是独立的、自给自足的，由相应的业务团队运营。Unit/Application 该模型允许为各种集群功能（集群AI/ML 、批处理、通用等）创建可重复使用的蓝图，并根据应用团队的要求出售集群。应用程序和平台团队都通过各自的GitOps存储库进行操作，以管理工作负载集群的部署。



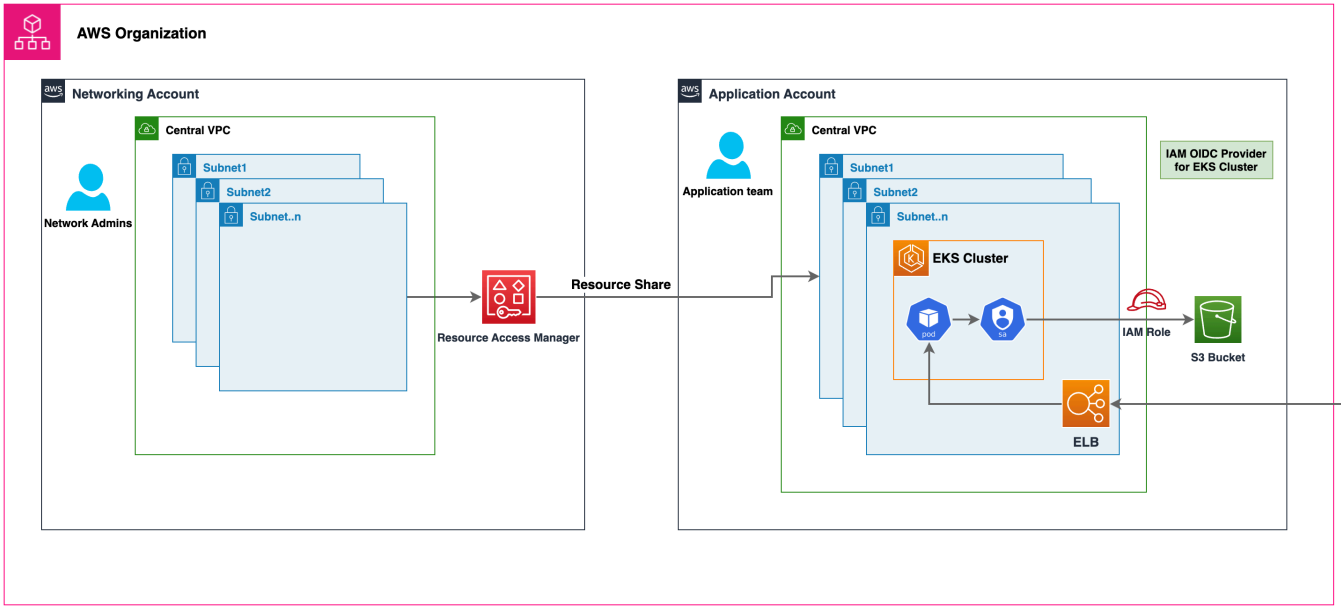

在上图中，Amazon EKS 集群和其他 AWS 资源部署到相应的工作负载账户。然后，在 EKS 容器中运行的工作负载使用 IRSA 或 EKS 容器身份访问其 AWS 资源。

GitOps 是一种管理应用程序和基础架构部署的方法，以便在 Git 存储库中以声明方式描述整个系统。它是一种操作模型，使您能够使用版本控制、不可变工件和自动化的最佳实践来管理多个 Kubernetes 集群的状态。在这种多集群模型中，每个工作负载集群都使用多个 Git 存储库进行引导，允许每个团队（应用程序、平台、安全等）在集群上部署各自的更改。

您将在每个账户中使用服务账户 (IRSA) 或 EKS Pod 身份的 IAM 角色来允许您的 EKS 工作负载获得临时的 aws 证书，从而安全地访问其他 AWS 资源。IAM 角色在相应的工作负载 AWS 账户中创建，并将其映射到 k8s 服务账户，以提供临时的 IAM 访问权限。因此，这种方法不需要跨账户访问权限。请按照服务账户的 IAM 角色文档了解如何在每个工作负载中为 IRSA 进行设置，以及有关如何在每个账户中设置 EKS 容器身份的 EKS Pod Identities 文档。

## 集中式联网

您还可以利用 AWS RAM 将 VPC 子网共享给工作负载账户，并在其中启动 Amazon EKS 集群和其他 AWS 资源。这支持集中式网络 managment/administration、简化的网络连接和去中心化的 EKS 集群。有关此方法的详细演练和注意事项，请参阅此 AWS 博客。




在上图中，AWS RAM 用于将子网从中央网络账户共享到工作负载账户。然后，EKS 集群和其他 AWS 资源将在相应的工作负载账户中的子网中启动。EKS 容器使用 IRSA 或 EKS 容器身份来访问其 AWS 资源。

## 集中集群与 De-centralized EKS 集群

决定使用集中式还是 De-centralized 将取决于您的要求。下表说明了每种策略的主要区别。

| # | 集中式 EKS 集群 | De-centralized EKS 集群 |
|---|---|---|
| 集群管理： | 管理单个 EKS 集群比管理多个集群更容易 | 为了减少管理多个 EKS 集群的运营开销，必须实现高效的集群管理自动化 |
| 成本效益： | 允许重复使用 EKS 集群和网络资源，从而提高成本效益 | 每个工作负载都需要网络和集群设置，这需要额外的资源 |
| 弹性： | 如果集群受损，集中式集群上的多个工作负载可能会受到影响 | 如果集群受损，则损害仅限于在该集群上运行的工作负载。所有其他工作负载均不受影响 |
| 隔离和安全： | Isolation/Soft Multi-tenancy 是使用 k8s 原生构造来实现的，比如。Namespaces 工 | 由于工作负载在不共享任何资源的单个集群和节点中运行，因此对计算资源进行更强的隔 |

| # | 集中式 EKS 集群 | De-centralized EKS 集群 |
| --- | --- | --- |
| | 作负载可能共享底层资源，例如 CPU、内存等。AWS 资源被隔离到自己的工作负载账户中，默认情况下，其他 AWS 账户无法访问这些账户。 | 离。AWS 资源被隔离到自己的工作负载账户中，默认情况下，其他 AWS 账户无法访问这些账户。 |
| 性能和可扩展性： | 随着工作负载增长到非常大的规模，您可能会在集群账户中遇到 kubernetes 和 AWS 服务配额。您可以部署其他集群账户以进一步扩展 | 随着集群和 VPC 的出现，每个工作负载都有更多可用 k8s 和 AWS 服务配额 |
| 联网： | 每个集群使用一个 VPC，从而简化了该集群上的应用程序的连接 | 必须在去中心化的 EKS 集群 VPC 之间建立路由 |
| Kubernetes 访问管理： | 需要在集群中维护许多不同的角色和用户，以便为所有工作负载团队提供访问权限并确保 kubernetes 资源得到适当隔离 | 简化了访问管理，因为每个集群都专用 workload/team |
| AWS 访问管理： | AWS 资源部署到他们自己的账户中，默认情况下，只能使用工作负载账户中的 IAM 角色进行访问。工作负载账户中的 IAM 角色是跨账户假设的，要么是 IRSA 或 EKS Pod 身份。 | AWS 资源部署到他们自己的账户中，默认情况下，只能使用工作负载账户中的 IAM 角色进行访问。工作负载账户中的 IAM 角色直接传送到具有 IRSA 或 EKS Pod 身份的 pod |

# 集群访问管理

有效的访问管理对于维护 Amazon EKS 集群的安全性和完整性至关重要。本指南探讨了 EKS 访问管理的各种选项，重点介绍如何使用 AWS IAM 身份中心（前身为 AWS SSO）。我们将比较不同的方法，讨论它们的权衡取舍，并重点介绍已知的局限性和注意事项。

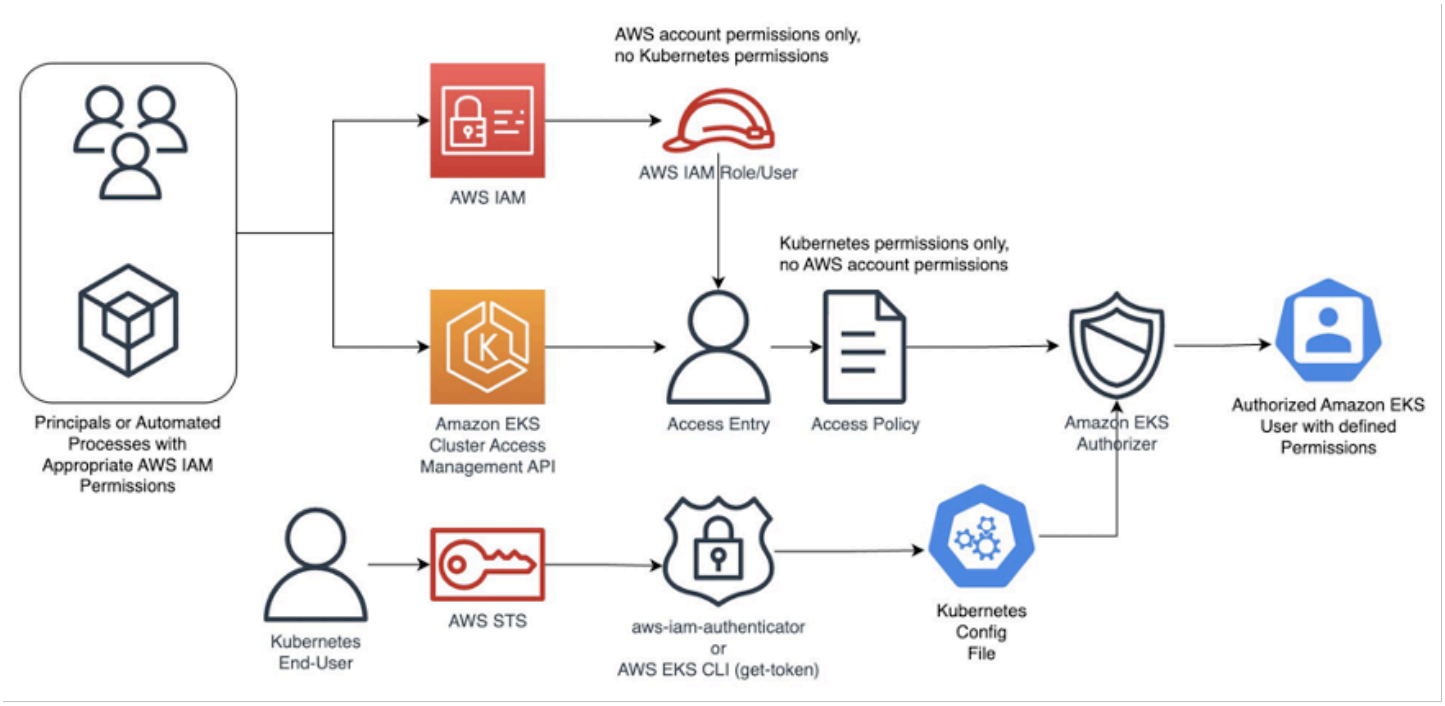
# EKS 访问管理选项

> **Note**
>
> ConfigMap-based 访问管理 (aws-auth ConfigMap) 已被弃用，取而代之的是集群访问管理 (CAM) API。对于新的 EKS 集群，请实施 CAM API 来管理集群访问权限。对于使用 aws-auth 的现有集群 ConfigMap，请迁移到使用 CAM API。

## 选项 1：带有集群访问管理 (CAM) API 的 AWS IAM 身份中心

- 集中式用户和权限管理
- 与现有身份提供商（例如 Microsoft AD、Okta 等）集成 PingId
- CAM API 使用访问条目将 AWS IAM 委托人（用户或角色）链接到 EKS 集群。这些条目与 IAM Identity Center 的托管身份配合使用，允许管理员控制在 Identity Center 中定义的用户和群组的集群访问权限。

### EKS 集群身份验证流程：




1. 委托人（人类用户）或自动流程通过出示相应的 AWS 账户权限通过 AWS IAM 进行身份验证。在此步骤中，它们会映射到相应的 AWS IAM 委托人（角色或用户）。

2. 接下来，通过定义适当的访问策略（仅包含 Kubernetes 权限），EKS 访问条目将此 IAM 委托人映射到 Kubernetes RBAC 委托人（用户或群组）。
3. 当 Kubernetes 最终用户尝试访问集群时，其身份验证请求将由 aws-iam-authenticator 或 AWS EKS CLI 处理，并根据 kubeconfig 文件中的集群环境进行验证。
4. 最后，EKS 授权者验证与经过身份验证的用户的访问条目相关的权限，并相应地授予或拒绝访问权限。
   - API 使用 Amazon EKS-specific 访问策略来定义每个访问条目的授权级别。这些策略可以映射到 IAM Identity Center 中设置的角色和权限，从而确保对 AWS 服务和 EKS 集群进行一致的访问控制。

## 与 ConfigMap-based 访问管理相比的优点：

1. 降低配置错误的风险：直接 API-based 管理可消除与手动 ConfigMap 编辑相关的常见错误。这有助于防止意外删除或语法错误，这些错误可能会将用户锁定在集群之外。
2. 增强的最低权限原则：不再需要集群创建者身份中的集群管理员权限，并允许进行更精细和适当的权限分配。您可以选择为破碎玻璃用例添加此权限。
3. 增强的安全模型：在应用访问条目之前提供对访问条目的内置验证。此外，还提供与 AWS IAM 更紧密的集成以进行身份验证。
4. 简化操作：通过 AWS-native 工具提供更直观的方式来管理权限。

## 最佳实践：

1. 使用 AWS Organizations 管理多个账户并应用服务控制策略 (SCP)。
2. 通过为不同的 EKS 角色（例如管理员、开发人员、只读）创建特定的权限集来实现最低权限原则。
3. 利用基于属性的访问控制 (ABAC)，根据用户属性为容器动态分配权限。
4. 定期审计和审查访问权限。

## Considerations/limitations:

1. Identity Center 生成的角色 ARN 具有随机后缀，因此很难在静态配置中使用。
2. Kubernetes 资源级别对细粒度权限的支持有限。自定义 Kubernetes RBAC 角色需要进行其他配置。除了 Kubernetes-native RBAC 之外，还可以考虑使用 Kyverno 在 EKS 集群中进行高级权限管理。

## 选项 2：AWS IAM Users/Roles 映射到 Kubernetes 群组

### 优点：

1. Fine-grained 控制 IAM 权限。
2. 可预测和静态角色 ARN

### 缺点：

1. 增加了用户账户的管理开销
2. 缺乏集中式身份管理
3. IAM 实体有可能激增

### 最佳实践：

1. 使用 IAM 角色代替 IAM 用户以提高安全性和可管理性
2. 为角色实施命名惯例，以确保一致性和易管理性
3. 利用 IAM 策略条件根据标签或其他属性限制访问权限。
4. 定期轮换访问密钥并查看权限。

### Considerations/limitations:

1. 管理大量用户或角色时出现可扩展性问题
2. 没有内置的单点登录功能

## 选项 3：OIDC 提供商

### 优点：

1. 与现有身份管理系统集成
2. 减少了用户帐户的管理开销

缺点：

1. 额外的配置复杂性
2. 身份验证期间的延迟可能会增加
3. 对外部身份提供商的依赖

最佳实践：

1. 仔细配置 OIDC 提供商，以确保安全令牌验证。
2. 使用短期令牌并实现令牌刷新机制。
3. 定期审核和更新 OIDC 配置。

查看本指南，了解将外部单一 Sign-On 提供商与 Amazon EKS 集成的参考实现方法

Considerations/limitations:

1. 与 IAM 相比，与 AWS 服务的本地集成有限。
2. OIDC 提供商的发行者 URL 必须可公开访问，EKS 才能发现签名密钥。

# 适用于工作负载的 AWS EKS Pod 身份与 IRSA

Amazon EKS 提供了两种向在 Amazon EKS 集群中运行的工作负载授予 AWS IAM 权限的方法：服务账户的 IAM 角色 (IRSA) 和 EKS Pod 身份。

虽然 IRSA 和 EKS Pod 身份都具有最低权限访问、凭据隔离和可审计性的优点，但 EKS Pod Identity 是向工作负载授予权限的推荐方式。

有关 EKS pod 的身份和凭证的详细指南，请参阅安全最佳实践中的身份和凭证部分。

# 建议

## 将 IAM 身份中心与 CAM API 相结合

- 简化管理：通过将集群访问管理 API 与 IAM Identity Center 配合使用，管理员可以管理 EKS 集群访问权限以及其他 AWS 服务，从而减少了在不同界面之间切换或 ConfigMaps 手动编辑的需求。

- 使用访问条目管理集群外的 IAM 主体的 Kubernetes 权限。您可以使用 EKS API、AWS 命令行界面、AWS 软件开发工具包、AWS 和 AWS 管理控制台来添加和管理对集群的访问权限。CloudFormation这意味着您可以使用与创建集群相同的工具来管理用户。
- 可以通过访问条目和访问策略将 Kubernetes 用户或群组映射到与 SSO 身份关联的 IAM 委托人，从而应用精细的 Kubernetes 权限。
- 首先，请按照更改身份验证模式使用访问条目，然后按照迁移现有的 aws-auth ConfigMap 条目来访问条目。

# 集群自动扩缩的最佳实践

本指南提供了有关集群自动扩缩的建议，包括自动模式、Karpenter 和 Kubernetes 集群自动扩缩器的指南。

主题


- EKS 自动模式
- Karpenter
- Cluster Autoscaler


## EKS 自动模式

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Amazon EKS 自动模式代表了 Kubernetes 基础设施管理的重大发展，它将安全且可扩展的集群基础设施与 AWS 管理的集成 Kubernetes 功能相结合。该服务提供完全托管的工作节点操作，客户无需设置托管节点组或 AutoScaling 组。

架构的关键区别在于，EKS Auto Mode 使用的 Karpenter-based 系统会自动预置 EC2 实例以响应 pod 请求。这些实例在 Bottlerocket AMI 上运行，预装了诸如 EBS CSI 驱动程序之类的附加组件，从而使基础设施真正由 AWS 管理。与传统的缩放方法相比：

- 传统的集群自动扩缩器 (CAS) 需要手动管理节点组，并且只能为每个节点组创建具有单一实例类型的节点
- Self-managed Karpenter 通过使用 EC2 队列 API 提供了更大的灵活性，并且可以配置不同的实例类型，但需要客户管理
- EKS 自动模式通过托管 NodePools 和自动处理所有扩展操作 NodeClasses

新系统引入了多项操作改进：

- Pod 驱动的自动扩展，无需手动配置节点组
- Built-in ALB/NLB 基于 Ingress 资源自动创建的托管负载平衡器控制器

- 集成的安全功能和预先配置的 Pod 身份
- 自动替换后，节点的最大运行时间为 21 天

从成本角度来看，EKS 自动模式保持标准 EC2 定价，同时仅为自动 Mode-managed 节点增加管理费。重要的是，客户仍然可以在同一个集群中混合使用自动模式托管节点和自我管理节点。

虽然 AWS 负责处理大部分运营方面，但客户仍负责集群版本管理，并且可以执行控制升级，从而触发工作节点的滚动更新。

## 使用自动模式的理由

自动模式面向那些希望获得 Kubernetes 和 EKS 的好处，但需要最大限度地减少 Kubernetes（如升级）和 installation/maintenance 关键平台部分（例如自动扩展、负载平衡和存储）的运营负担的用户。自动模式让 EKS 更进一步，最大限度地减少了 Kubernetes 维护带来的无差别繁重的工作

## 常见问题解答

### EKS 自动模式和开源 Karpenter 有什么区别？

EKS 自动模式是一套大型功能，可让运行生产级 Kubernetes 变得简单。其中一项功能是完全托管的 Karpenter 的自动缩放优势。从操作的角度来看，唯一的区别在于在 EKS 自动模式下，你不需要自己管理 Karpenter pod 的部署、扩展和升级。所有其他操作，例如托管操作，其 NodePools 工作原理 NodeClasses与开源 Karpenter 相同。

### 我能否与 Auto 节点一起运行托管 Mode-managed 节点组？

是的，您可以通过托管节点组运行静态节点，同时还可以在自动模式下提供的自动缩放节点旁边运行

### 我能否将集群从标准 EKS 迁移到 EKS 自动模式？

是的，可以在官方 AWS 文档中找到在现有集群上启用 EKS 自动模式的说明

注意事项：1. 启用自动模式后，您需要卸载已安装的所有现在由自动模式管理的组件，例如 Karpenter 或 AWS Load Balancer Controller 2。你需要确保你安装的插件是最新的。参见文档。

### 如何 NodePools 在 EKS 自动模式下进行配置？

新集群将预先配置两个 NodePools

一般用途

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: general-purpose
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      terminationGracePeriod: 24h0m0s
```

这 NodePool 会指示 Karpenter 启动具有以下特征的节点：

1. “按需” 的容量类型
2. C、M 或 R 的实例类型
3. 4 的实例生成
4. AMD 架构
5. Linux OS

它还通过声明在任何给定时间所有节点中只有10%可能处于中断状态，并且只有在节点为空或未充分利用时才应进行整合，从而定义了缩小逻辑。

系统

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: system
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
            - arm64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      taints:
        - effect: NoSchedule
```

NodePool 这与 “通用” 类似，但有以下区别：

1. 它允许节点采用 ARM 架构和 AMD 架构
2. 除非可以容忍 “”CriticalAddonsOnly， NoSchedule 否则它会用污染这些节点。这是供 EKS 插件内部使用的

自定义

您可以 NodePools 根据需要创建自己的自定义。要了解更多信息，NodePools 请查阅 Karpenter 文档。

## 启动新节点时，我能否自定义自动模式使用的 AMI？

不，目前唯一支持的 AMI 是 Bottlerocket Amazon-provided

## 如何在我的 Kubernetes 主机上安装自定义工具或代理？

由于不支持 AMI 自定义，因此如果您需要主机级软件来执行安全扫描等操作，则应将工作负载部署为 Kubernetes。DaemonSet

## 当我配置新的 EKS Auto Mode 集群时，我的集群数据平面中正在运行哪些组件？

如果您使用 eksctl 或 AWS 控制台创建集群，则唯一在 EKS 自动模式集群中运行的 pod 是 Kubernetes Metrics Server 容器。EKS 自动模式的其他组件，例如 Karpenter、AWS Load Balancer Controller 和 EBS CSI 驱动程序，都在集群外运行和管理。

## 正在运行哪些托管组件来支持我的新 EKS 自动模式集群？

EKS 自动模式可完全自动部署生产级 Kubernetes 所需的大部分数据平面。这包括：

- Karpenter，用于自动缩放集群的计算
- AWS Load Balancer Controller 允许您通过自动的 Elastic Load Balancer 集成轻松公开 Kubernetes 服务
- VPC CNI
- 集群 DNS
- kube-proxy
- EBS CSI
- EKS 容器组身份代理

- EKS 节点监控代理

### 如何对以前在我的集群中作为 pod 运行的自动模式组件进行故障排除？

使用 EKS 自动模式，AWS Load Balancer Controller 和 Karpenter 等许多组件都是在集群之外为您管理的，因此您无法像自我管理时那样查看日志。如果您需要对自动模式功能的功能进行故障排除，请创建 AWS Support Ticket。

## Karpenter

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Karpenter 是一个开源项目，旨在增强 Kubernetes 集群中的节点生命周期管理。它可以根据 Pod 的特定调度需求自动配置和取消配置节点，从而实现高效的扩展和成本优化。它的主要功能是：

- 监控 Kubernetes 调度器由于资源限制而无法调度的容器。
- 评估不可调度 Pod 的调度要求（资源请求、节点选择器、亲和力、容忍度等）。
- 配置满足这些 Pod 要求的新节点。
- 不再需要节点时将其移除。

使用 Karpenter，您可以定义 NodePools 对节点配置的限制，例如污点、标签、要求（实例类型、区域等），以及对总预配置资源的限制。部署工作负载时，您可以在 Pod 规格中指定各种调度限制，例如资源 requests/limits、节点选择器、 node/pod 亲和力、容忍度和拓扑分布限制。然后，Karpenter 将根据这些规格配置大小合适的节点。

使用 Karpenter 的理由

在 Karpenter 推出之前，Kubernetes 用户主要依靠 Amazon EC2 Auto Scaling 组和 Kubernetes 集群自动扩缩器 (CAS) 来动态调整其集群的计算容量。使用 Karpenter，您无需创建数十个节点组即可实现 Karpenter 所获得的灵活性和多样性。与 CAS 不同，Karpenter 与 Kubernetes 版本的耦合不那么紧密，也不需要你在 AWS 和 Kubernetes API 之间切换。

Karpenter 将实例编排职责整合到一个系统中，该系统更简单、更稳定且支持集群。Karpenter 旨在通过提供简化的方法来克服集群自动扩缩器带来的一些挑战：

- 根据工作负载要求配置节点。
- 使用灵活的NodePool 选项，按实例类型创建不同的节点配置。Karpenter 可以让您通过单个灵活的方式管理不同的工作负载容量，而不是管理许多特定的自定义节点组。 NodePool
- 通过快速启动节点和调度 Pod，实现大规模改进 Pod 调度。

有关使用 Karpenter 的信息和文档，请访问 karpenter.sh 网站。

## 建议

最佳实践分为关于 Karpenter 本身和 pod 调 NodePools度的部分。

## Karpenter 最佳实践

以下最佳实践涵盖了与 Karpenter 本身相关的主题。

### 锁定生产集群中的 AMI

我们强烈建议您将 Karpenter 用于生产集群的知名亚马逊系统映像 (AMI) 进行固定。将别名设置为`@latest`，或者使用其他方法导致在发布未经测试的 AMI 时部署它们，会给生产集群带来工作负载故障和停机的风险。`amiSelector`因此，我们强烈建议您在非生产集群中测试较新版本时，为您的生产集群固定经过测试的 AMI 工作版本。例如，您可以按 NodeClass 如下方式在中设置别名：

```
amiSelectorTerms
  - alias: al2023@v20240807
```

有关在 Karpenter 中管理和锁定 AMI 的信息，请参阅 Karpenter 文档中的管理 AMI。

### 使用 Karpenter 处理容量需求不断变化的工作负载

与自动扩缩组 (ASG) 和托管节点组 (MNG) 相比，Karpenter 使扩展管理更接近 Kubernetes 原生 API。ASG 和 MNG 是 AWS-native 抽象概念，其中扩展是基于 AWS 级别的指标（例如 EC2 CPU 负载）触发的。Clu@@ ster Autoscaler 将 Kubernetes 抽象与 AWS 抽象联系起来，但因此失去了一些灵活性，例如为特定可用区调度。

Karpenter 删除了一层 AWS 抽象，将一些灵活性直接带入 Kubernetes 中。Karpenter 最适合工作负载处于高峰期、高峰期或计算要求不同的集群。MNG 和 ASG 非常适合运行往往更加静态和一致的工作负载的集群。根据您的要求，您可以混合使用动态和静态管理的节点。

### 当... 时，可以考虑其他自动缩放项目

你需要在 Karpenter 中仍在开发的功能。由于 Karpenter 是一个相对较新的项目，因此如果您需要尚未包含在 Karpenter 中的功能，请暂时考虑其他自动缩放项目。

### 在 EKS Fargate 或属于节点组的工作节点上运行 Karpenter 控制器

Karpenter 是使用 Helm 图表安装的。Helm 图表将 Karpenter 控制器和 webhook pod 安装为部署，需要在控制器可用于扩展集群之前运行该部署。我们建议至少有一个包含至少一个工作节点的小型节点组。或者，你可以通过为命名空间创建 Fargate 配置文件在 EKS Fargate 上运行这些 pod。`karpenter`这样做会导致部署到这个命名空间中的所有 pod 都在 EKS Fargate 上运行。不要在由 Karpenter 管理的节点上运行 Karpenter。

### Karpenter 不支持自定义启动模板

v1 API 不支持自定义启动模板。您可以使用自定义用户数据 and/or 直接在中指定自定义 AMI。EC2NodeClass有关如何执行此操作的更多信息，请访问NodeClasses。

### 排除不适合您的工作负载的实例类型

如果集群中运行的工作负载不需要特定实例类型，请考虑使用`node.kubernetes.io/instance-type`密钥排除这些实例类型。

以下示例显示了如何避免配置大型 Graviton 实例。

```
- key: node.kubernetes.io/instance-type
  operator: NotIn
  values:
  - m6g.16xlarge
  - m6gd.16xlarge
  - r6g.16xlarge
  - r6gd.16xlarge
  - c6g.16xlarge
```

### 使用 Spot 时启用中断处理

Karpenter 支持原生中断处理，可以处理非自愿中断事件，例如竞价型实例中断、计划维护事件、可能中断工作负载的实例termination/stopping 事件。当 Karpenter 检测到节点的此类事件时，它会自动提前污染、清空和终止受影响的节点，以便在中断之前开始优雅地清理工作负载。如果 Spot 中断，提醒 2 分钟，Karpenter 会快速启动一个新节点，这样就可以在回收实例之前移动 Pod。要启用中断处

理，请使用为此目的配置的 SQS 队列的名称来配置 `--interruption-queue` CLI 参数。不建议将 Karpenter 中断处理与节点终止处理程序一起使用，如下所述。

需要检查点或其他形式的优雅排水、需要在关闭前 2 分钟的 Pod 应在集群中启用 Karpenter 中断处理。

## 没有出站互联网访问权限的 Amazon EKS 私有集群

将 EKS 集群配置到没有互联网路由的 VPC 时，必须确保已根据 EKS 文档中显示的私有集群要求配置环境。此外，您需要确保已在您的 VPC 中创建了 STS VPC 区域终端节点。否则，您将看到与下面显示的错误相似的错误。

```
{"level":"FATAL","time":"2024-02-29T14:28:34.392Z","logger":"controller","message":"Checking
 EC2 API connectivity, WebIdentityErr: failed to retrieve credentials
\ncaused by: RequestError: send request failed\ncaused by: Post
 \"https://sts.<region>.amazonaws.com/\": dial tcp 54.239.32.126:443: i/o
 timeout","commit":"596ea97"}
```

这些更改在私有集群中是必要的，因为 Karpenter Controller 使用服务账户 (IRSA) 的 IAM 角色。配置了 IRSA 的 Pod 通过调用 AWS Security Token Service (AWS STS) API 获取证书。如果没有出站互联网接入，则必须在 VPC 中创建和使用 AWS STS VPC 终端节点。

私有集群还要求您为 SSM 创建 VPC 终端节点。当 Karpenter 尝试配置新节点时，它会查询启动模板配置和 SSM 参数。如果您的 VPC 中没有 SSM VPC 终端节点，则会导致以下错误：

```
{"level":"ERROR","time":"2024-02-29T14:28:12.889Z","logger":"controller","message":"Unable
 to hydrate the AWS launch template cache, RequestCanceled: request context canceled
\ncaused by: context canceled","commit":"596ea97","tag-key":"karpenter.k8s.aws/
cluster","tag-value":"eks-workshop"}
...
{"level":"ERROR","time":"2024-02-29T15:08:58.869Z","logger":"controller.nodeclass","message":"d
 amis from ssm, getting ssm parameter \"/aws/service/eks/optimized-ami/1.27/amazon-
linux-2/recommended/image_id\", RequestError: send request failed\ncaused by:
 Post \"https://ssm.<region>.amazonaws.com/\": dial tcp 67.220.228.252:443: i/o
 timeout","commit":"596ea97","ec2nodeclass":"default","query":"/aws/service/eks/
optimized-ami/1.27/amazon-linux-2/recommended/image_id"}
```

价目表查询 API 没有 VPC 终端节点。因此，定价数据将随着时间的推移而过时。Karpenter通过在其二进制文件中包含按需定价数据来解决这个问题，但只有在Karpenter升级时才会更新这些数据。定价数据请求失败将导致出现以下错误消息：

```
{"level":"ERROR","time":"2024-02-29T15:08:58.522Z","logger":"controller.pricing","message":"ret
 on-demand pricing data, RequestError: send request failed\ncaused by: Post
 \"https://api.pricing.<region>.amazonaws.com/\": dial tcp 18.196.224.8:443:
 i/o timeout; RequestError: send request failed\ncaused by: Post \"https://
api.pricing.<region>.amazonaws.com/\": dial tcp 18.185.143.117:443: i/o
 timeout","commit":"596ea97"}
```

要在完全私有的 EKS 集群中使用 Karpenter，并了解要创建哪些 VPC 终端节点，请参阅此文档。

## 正在创建 NodePools

以下最佳实践涵盖了与创建相关的主题 NodePools。

### 在... NodePools 时创建多个

当不同的团队共享一个集群并需要在不同的工作节点上运行其工作负载，或者对操作系统或实例类型有不同的要求时，可以创建多个集群 NodePools。例如，一个团队可能想使用 Bottlerocket，而另一个团队可能想要使用 Amazon Linux。同样，一个团队可能可以使用昂贵的 GPU 硬件，而另一个团队不需要这些硬件。使用多个 NodePools 可以确保每个团队都可以使用最合适的资产。

### 相互排 NodePools 斥或加权创作

NodePools 建议创建互斥或加权的，以提供一致的调度行为。如果它们不匹配并且 NodePools 有多个匹配，Karpenter 将随机选择使用哪个，从而导致意想不到的结果。创建多个有用的示例 NodePools 包括：

NodePool 使用 GPU 创建并仅允许特殊工作负载在以下（昂贵的）节点上运行：

```
# NodePool for GPU Instances with Taints
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu
spec:
  disruption:
    consolidateAfter: 1m
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    metadata: {}
    spec:
      nodeClassRef:
```

```
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - p3.8xlarge
        - p3.16xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
      taints:
      - effect: NoSchedule
        key: nvidia.com/gpu
        value: "true"
```

在可容忍污点的情况下部署：

```
# Deployment of GPU Workload will have tolerations defined
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inflate-gpu
spec:
    spec:
      tolerations:
      - key: "nvidia.com/gpu"
        operator: "Exists"
        effect: "NoSchedule"
```

对于其他团队的常规部署，该 NodePool 规范可能包括 NodeAffinity。然后，部署可以使用节点 SelectorTerms 进行匹配`billing-team`。

```
# NodePool for regular EC2 instances
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: generalcompute
spec:
  template:
    metadata:
      labels:
        billing-team: my-team
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - m5.large
        - m5.xlarge
        - m5.2xlarge
        - c5.large
        - c5.xlarge
        - c5a.large
        - c5a.xlarge
        - r5.large
        - r5.xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
```

使用 NodeAffinity 进行部署：

```
# Deployment will have spec.affinity.nodeAffinity defined
kind: Deployment
metadata:
  name: workload-my-team
spec:
  replicas: 200
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "billing-team"
                  operator: "In"
                  values: ["my-team"]
```

### 使用计时器 (TTL) 自动从集群中删除节点

您可以在已配置的节点上使用计时器来设置何时删除没有工作负载 Pod 或已达到过期时间的节点。节点过期可用作升级手段，这样节点就会停用并替换为更新的版本。有关使用`spec.template.spec`配置节点到期的信息，请参阅 Karpenter 文档中的过期时间。

### 避免过度限制 Karpenter 可以配置的实例类型，尤其是在使用 Spot 时

使用 Spot 时，Karpenter 使用价格容量优化的分配策略来配置 EC2 实例。此策略指示 EC2 根据您正在启动的实例数量从最深的池中配置实例，并且中断风险最低。然后，EC2 队列请求这些池中价格最低的竞价型实例。您允许 Karpenter 使用的实例类型越多，EC2 就越能更好地优化您的竞价型实例的运行时间。默认情况下，Karpenter 将使用 EC2 在您的集群部署所在的区域和可用区中提供的所有实例类型。Karpenter 会根据待处理的 pod 从所有实例类型中进行智能选择，以确保您的 Pod 被调度到大小适当、配备合适的实例上。例如，如果您的容器不需要 GPU，Karpenter 就不会将您的容器调度到支持 GPU 的 EC2 实例类型。当您不确定要使用哪些实例类型时，可以运行 Amazon ec2-instance-selector 来生成符合您的计算要求的实例类型列表。例如，CLI 将内存 vCPU、架构和区域作为输入参数，并为您提供满足这些限制的 EC2 实例列表。

```
$ ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 -r ap-
southeast-1
c5.large
c5a.large
c5ad.large
c5d.large
```

```
c6i.large
t2.medium
t3.medium
t3a.medium
```

在使用 Spot 实例时，不应对 Karpenter 设置太多限制，因为这样做会影响应用程序的可用性。例如，假设特定类型的所有实例都被回收了，没有合适的替代方案可以替换它们。在配置的实例类型的竞价容量补充完毕之前，您的 Pod 将保持待处理状态。您可以通过将实例分散到不同的可用区来降低容量不足错误的风险，因为各个可用区的竞价池不同。也就是说，一般的最佳做法是允许 Karpenter 在使用 Spot 时使用不同的实例类型。

## 调度 Pod

以下最佳实践与使用 Karpenter 进行节点配置的集群中部署 pod 有关。

### 遵循 EKS 最佳实践以实现高可用性

如果您需要运行高可用性应用程序，请遵循一般的 EKS 最佳实践建议。有关如何跨节点和区域分布 Pod 的详细信息，请参阅 Karpenter 文档中的拓扑分布。使用中断预算来设置需要维护的最低可用容量，以防有人试图驱逐或删除 Pod。

### 使用分层约束限制云提供商提供的计算功能

Karpenter 的分层约束模型允许您创建一组复杂的 pod 部署约束，以获得最匹配的 pod 调度。NodePool Pod 规范可以请求的约束条件示例包括以下内容：

- 需要在只有特定应用程序可用的可用区中运行。例如，假设您的 pod 必须与在位于特定可用区的 EC2 实例上运行的另一个应用程序通信。如果您的目标是减少 VPC 中的跨可用区流量，则可能需要将 Pod 托管在 EC2 实例所在的可用区中。这种定位通常是使用节点选择器完成的。有关节点选择器的更多信息，请参阅 Kubernetes 文档。
- 需要某些类型的处理器或其他硬件。有关要求 pod 在 GPU 上运行的 pod 规格示例，请参阅 Karpenter 文档的加速器部分。

### 创建账单警报以监控您的计算支出

当您将集群配置为自动扩展时，您应该创建账单警报，以便在支出超过阈值时向您发出警告，并在 Karpenter 配置中添加资源限制。使用 Karpenter 设置资源限制与设置 AWS 自动扩展组的最大容量类似，因为它代表了 Karpenter 可以实例化的最大计算资源量。NodePool

> **Note**
>
> 无法为整个集群设置全局限制。限制适用于特定 NodePools。

下面的片段告诉 Karpenter 最多只能配置 1000 个 CPU 内核和 1000Gi 的内存。只有在达到或超过限制时，Karpenter 才会停止添加容量。当超过限制时，Karpenter 控制器将在控制器的日志中写入`memory resource usage of 1001 exceeds limit of 1000`或类似的消息。如果您要将容器日志路由到 CloudWatch 日志，则可以创建指标筛选器来查找日志中的特定模式或术语，然后创建警报，在您配置的指标阈值被突破时提醒您。CloudWatch

有关使用 Karpenter 限制的更多信息，请参阅 Karpenter 文档中的设置资源限制。

```
spec:
  limits:
    cpu: 1000
    memory: 1000Gi
```

如果您不使用限制或限制 Karpenter 可以配置的实例类型，Karpenter 将继续根据需要向您的集群添加计算容量。虽然以这种方式配置 Karpenter 可以让您的集群自由扩展，但也可能带来显著的成本影响。正是出于这个原因，我们建议配置账单警报。账单警报允许您在账户中计算出的预估费用超过定义的阈值时收到提醒和主动通知。有关更多信息，请参阅设置 Amazon CloudWatch 账单警报以主动监控预估费用。

您可能还需要启用成本异常检测，这是一项 AWS 成本管理功能，它使用机器学习来持续监控您的成本和使用情况，以检测异常支出。更多信息可在 AWS 成本异常检测入门指南中找到。如果您已经在 AWS Budgets 中创建预算，还可以配置一项操作，以便在突破特定阈值时通知您。通过预算操作，您可以发送电子邮件、向 SNS 主题发布消息或向 Slack 等聊天机器人发送消息。有关更多信息，请参阅配置 AWS Budgets 操作。

### 使用木匠。 sh/do-not-drustrup 注释以防止 Karpenter 取消配置节点

如果您在 Karpenter-provisioned节点上运行关键应用程序，例如长时间运行的批处理作业或有状态的应用程序，并且该节点的 TTL 已过期，则该应用程序将在实例终止时中断。通过向 Pod 添加`karpenter.sh/do-not-disrupt`注解，您就是在指示 Karpenter 保留该节点，直到 Pod 终止或`karpenter.sh/do-not-disrupt`注释被移除。有关更多信息，请参阅 Distruption 文档。

如果节点上唯一剩下的非守护程序集容器是与任务关联的 pod，那么只要任务状态为成功或失败，Karpenter 就可以瞄准和终止这些节点。

### 使用整合时，为所有非 CPU 资源配置 requests=limits

整合和调度通常是通过比较节点上的 pod 资源请求和可分配的资源量来进行的。不考虑资源限制。例如，内存限制大于内存请求的 Pod 可能会突增到请求之上。如果同一节点上的多个 Pod 同时爆裂，则可能导致某些 Pod 因内存不足 (OOM) 情况而终止。整合可以使这种情况更有可能发生，因为只有考虑节点的请求才能将 pod 打包到节点上。

### LimitRanges 用于配置资源请求和限制的默认值

由于 Kubernetes 不设置默认请求或限制，因此容器对底层主机、CPU 和内存的资源消耗不受限制。Kubernetes 调度器会查看 Pod 的总请求数（来自 Pod 容器的请求总数或容器的 Init 容器的总资源中较高者），以确定将 Pod 调度到哪个工作节点上。同样，Karpenter 会考虑 pod 的请求，以确定其提供的实例类型。你可以使用限制范围为命名空间应用合理的默认值，以防某些 pod 未指定资源请求。

请参见为命名空间配置默认内存请求和限制

### 将准确的资源请求应用于所有工作负载

当有关您的工作负载要求的信息准确时，Karpenter 能够启动最适合您的工作负载的节点。如果使用 Karpenter 的合并功能，这一点尤其重要。

请参见为所有工作负载配置和调整资源 Requests/Limits 规模

## CoreDNS 推荐

### 更新 CoreDNS 的配置以保持可靠性

在由 Karpenter 管理的节点上部署 CoreDNS 容器时，鉴于 Karpenter 在 terminating/creating 快速新建节点中具有动态特性以适应需求，因此建议遵循以下最佳实践：

CoreDNS lameduck 时长

CoreDNS 就绪情况调查

这将确保 DNS 查询不会被定向到尚未准备就绪或已终止的 CoreDNS Pod。

## Karpenter 蓝图

由于 Karpenter 采用应用程序优先的方法为 Kubernetes 数据平面配置计算容量，因此您可能想知道如何正确配置常见的工作负载场景。Karpenter Blueprin ts 是一个存储库，其中包含遵循此处描述的最佳

实践的常见工作负载场景列表。您将拥有所需的所有资源，甚至可以创建配置了 Karpenter 的 EKS 集群，并测试存储库中包含的每个蓝图。您可以组合不同的蓝图，最终创建工作负载所需的蓝图。

### 其他资源

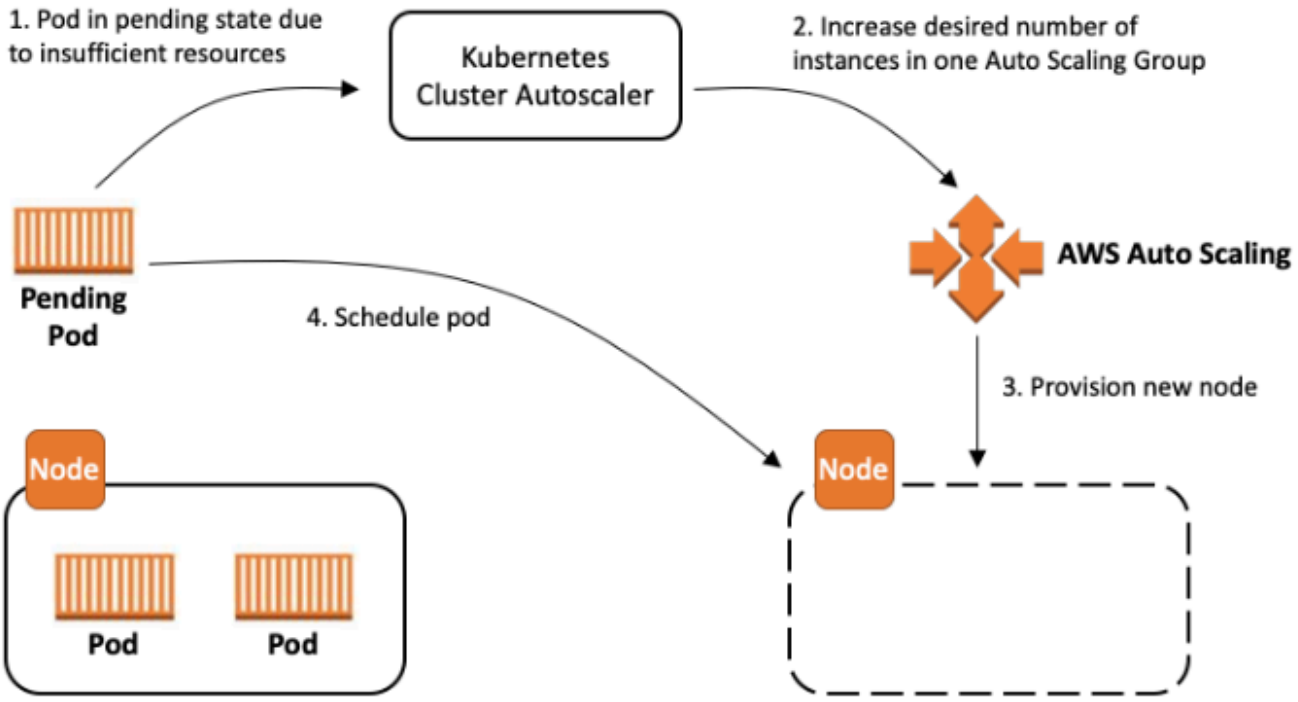
- Karpenter 沉浸式日工作坊
- Karpenter 成本优化研讨会
- EKS 研讨会——Karpenter
- Karpenter vs 集群自动扩缩器
- 2023 年 re: Invent 上的 Karpenter 会议
- 教程：使用 Amazon EC2 Spot 和 Karpenter 以更低的价格运行 Kubernetes 集群

## Cluster Autoscaler

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

### 概述

Kubernetes 集群自动扩缩器是 SIG Autoscal ing 维护的流行集群自动扩缩解决方案。它负责确保您的集群有足够的节点来调度 Pod，而不会浪费资源。它会监视无法调度的 Pod 以及未充分利用的节点。然后，它会模拟节点的添加或移除，然后再将更改应用于您的集群。集群自动扩缩器中的 AWS 云提供商实现控制您的 EC2 Auto Scaling 组的`.DesiredReplicas`字段。




本指南将提供一个心理模型，用于配置集群自动扩缩程序并选择一组最佳的权衡方案来满足您的组织需求。虽然没有单一的最佳配置，但有一组配置选项可以让您在性能、可扩展性、成本和可用性之间进行权衡。此外，本指南还将提供有关优化 AWS 配置的提示和最佳实践。

## 术语表

本文档中将经常使用以下术语。这些术语可能具有广泛的含义，但仅限于本文档的以下定义。

可扩展性是指随着 Kubernetes 集群的 Pod 和节点数量的增加，集群自动扩缩程序的表现如何。当达到可扩展性限制时，集群自动扩缩程序的性能和功能就会降低。当集群自动扩缩器超出其可扩展性限制时，它可能无法再在您的集群中添加或删除节点。

性能是指集群自动扩缩器能够以多快的速度做出和执行扩展决策。性能完美的 Cluster Autoscaler 会立即做出决定并触发缩放动作以响应刺激，例如 pod 变得不可调度。

可用性意味着可以快速、无中断地调度 pod。这包括何时需要对新创建的 Pod 进行调度，以及缩小规模的节点何时终止调度给它的所有剩余 Pod。

成本由事件中扩展和扩展背后的决策决定。如果现有节点未得到充分利用，或者添加的新节点对于传入的 Pod 来说太大，就会浪费资源。根据用例的不同，由于做出激进的缩减决定，过早终止 pod 可能会产生成本。

节点组是一个抽象的 Kubernetes 概念，用于集群中的一组节点。它不是真正的 Kubernetes 资源，而是作为抽象存在于集群自动扩缩器、集群 API 和其他组件中。节点组中的节点共享标签和污点等属性，但可能由多个可用区或实例类型组成。

EC2 Auto Scaling 组可用作 EC2 上节点组的实现。EC2 Auto Scaling 组配置为启动自动加入其 Kubernetes 集群的实例，并在 Kubernetes API 中为其相应的节点资源应用标签和污点。

EC2 托管节点组是 EC2 上节点组的另一种实现。它们消除了手动配置 EC2 AutoScaling Group 的复杂性，并提供了额外的管理功能，例如节点版本升级和优雅的节点终止。

## 操作集群自动扩缩器

Cluster Autoscaler 通常作为部署安装在集群中。它使用领导者选举来确保高可用性，但一次只能通过一个副本完成工作。它不能横向扩展。对于基本设置，默认设置应该使用提供的安装说明开箱即用，但有几点需要记住。

请确保：

- 集群自动扩缩程序的版本与集群的版本相匹配。跨版本兼容性未经过测试或支持。
- 除非您有阻止使用此模式的特定高级用例，否则将启用@@ 自动发现。

## 使用 IAM 角色的最低特权访问权限

使用 Auto Discovery 时，我们强烈建议您通过限制操作`autoscaling:SetDesiredCapacity`和`autoscaling:TerminateInstanceInAutoScalingGroup`围限于当前集群的 Auto Scaling 组来使用最低权限访问权限。

这将防止在一个集群中运行的集群 Autoscaler 修改另一个集群中的节点组（例如），即使`--node-group-auto-discovery`参数的范围未缩小到使用标签的集群节点组。`k8s.io/cluster-autoscaler/<cluster-name>`

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:SetDesiredCapacity",
                "autoscaling:TerminateInstanceInAutoScalingGroup"
            ],
```

```
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/enabled": "true",
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/my-cluster": "owned"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:DescribeAutoScalingInstances",
                "autoscaling:DescribeLaunchConfigurations",
                "autoscaling:DescribeScalingActivities",
                "autoscaling:DescribeTags",
                "ec2:DescribeImages",
                "ec2:DescribeInstanceTypes",
                "ec2:DescribeLaunchTemplateVersions",
                "ec2:GetInstanceTypesFromInstanceRequirements",
                "eks:DescribeNodegroup"
            ],
            "Resource": "*"
        }
    ]
}
```

## 配置您的节点组

有效的自动扩缩首先要为集群正确配置一组节点组。选择正确的节点组集是最大限度地提高可用性和降低工作负载成本的关键。AWS 使用 EC2 Auto Scaling 组来实现节点组，这些组可以灵活应用于大量用例。但是，集群自动扩缩器会对您的节点组进行一些假设。使您的 EC2 Auto Scaling 组配置与这些假设保持一致可以最大限度地减少不良行为。

请确保：

- 节点组中的每个节点都有相同的调度属性，例如标签、污点和资源。
  - 对于 MixedInstancePolicies，CPU、内存和 GPU 的实例类型的形状必须相同
  - 策略中指定的第一个实例类型将用于模拟调度。
  - 如果您的策略有其他实例类型和更多资源，则在扩展后可能会浪费资源。
  - 如果您的策略中包含其他实例类型，且资源较少，则 Pod 可能无法在实例上调度。

- 与具有较少节点的许多节点组相比，具有多个节点的节点组更受青睐。这将对可扩展性产生最大的影响。
- 如果两个系统都提供支持，请尽可能首选 EC2 功能（例如区域 MixedInstancePolicy）

> **Note**
>
> 我们建议使用 EKS 托管节点组。托管节点组具有强大的管理功能，包括集群自动扩缩器的功能，例如自动发现 EC2 Auto Scaling 组和优雅地终止节点。

## 针对性能和可扩展性进行优化

了解自动扩缩算法的运行时复杂性将有助于您调整集群自动扩缩器，使其在节点数超过 1,000 的大型集群中继续平稳运行。

调整集群自动扩缩器可扩展性的主要旋钮是提供给进程的资源、算法的扫描间隔以及集群中节点组的数量。该算法的真实运行时复杂性还涉及其他因素，例如调度插件的复杂性和 pod 的数量。这些参数被认为是不可配置的参数，因为它们是集群工作负载的自然参数，并且不容易进行调整。

集群自动扩缩器将整个集群的状态加载到内存中，包括 Pod、节点和节点组。在每个扫描间隔内，该算法都会识别不可调度的 pod，并模拟每个节点组的调度。调整这些因素会带来不同的权衡取舍，对于您的用例，应仔细考虑这些权衡。

### 垂直自动缩放集群自动扩缩器

将集群自动扩缩器扩展到更大的集群的最简单方法是增加其部署的资源请求。对于大型集群，应增加内存和 CPU，尽管这会因集群大小而有很大差异。自动缩放算法将所有 pod 和节点存储在内存中，在某些情况下，这可能会导致内存占用量大于 1 GB。增加资源通常是手动完成的。如果你发现持续的资源调整会带来运营负担，可以考虑使用 Addon Resizer 或 Vertic al Pod Autoscaler。

### 减少节点组的数量

尽量减少节点组的数量是确保集群自动扩缩程序在大型集群上继续保持良好性能的一种方法。对于某些按团队或应用程序构建节点组的组织来说，这可能具有挑战性。虽然 Kubernetes API 完全支持这一点，但这被认为是一种集群自动扩缩程序的反模式，会对可扩展性产生影响。使用多个节点组（例如 Spot 或 GPU）的原因有很多，但在许多情况下，还有其他设计可以在使用少量组的情况下实现相同的效果。

请确保：

- Pod 隔离是使用命名空间而不是节点组完成的。
  - 在低信任度多租户集群中，这可能不可能实现。
  - Pod ResourceRequests 和 ResourceLimits 已正确设置以避免资源争用。
  - 更大的实例类型将带来更优的垃圾箱打包效果并减少系统容器开销。
- NodeTaints 或 NodeSelectors 用于将 pod 作为例外情况进行调度，而不是通常使用。
- 区域资源被定义为具有多个可用区的单个 EC2 Auto Scaling 组。

## 缩短扫描间隔

低扫描间隔（例如 10 秒）将确保集群自动扩缩程序在 pod 变得不可调度时尽快做出响应。但是，每次扫描都会导致对 Kubernetes API 和 EC2 Auto Scaling Group 或 EKS 托管节点组 API 进行许多 API 调用。这些 API 调用可能会导致 Kubernetes 控制平面出现速率限制甚至服务不可用。

默认扫描间隔为 10 秒，但在 AWS 上，启动节点需要更长的时间才能启动新实例。这意味着，可以在不显著增加整体纵向扩展时间的情况下提高扫描间隔时间。例如，如果启动节点需要 2 分钟，则将间隔更改为 1 分钟，则需要权衡 API 调用减少 6 倍，而扩展速度降低 38%。

## 跨节点组分片

可以将集群自动扩缩器配置为在一组特定的节点组上运行。使用此功能，可以部署集群自动扩缩器的多个实例，每个实例都配置为在不同的节点组上运行。这种策略允许您使用任意数量的节点组，用成本换取可扩展性。我们只建议将其作为提高性能的最后手段。

Cluster Autoscaler 最初不是为此配置设计的，因此会产生一些副作用。由于分片无法通信，因此多个自动扩缩器可能会尝试调度不可调度的 pod。这可能会导致对多个节点组进行不必要的扩展。这些额外的节点将在之后缩减规模`scale-down-delay`。

```
metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-1

...

--nodes=1:10:k8s-worker-asg-1
--nodes=1:10:k8s-worker-asg-2
```

```
---

metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-2

...

--nodes=1:10:k8s-worker-asg-3
--nodes=1:10:k8s-worker-asg-4
```

请确保：

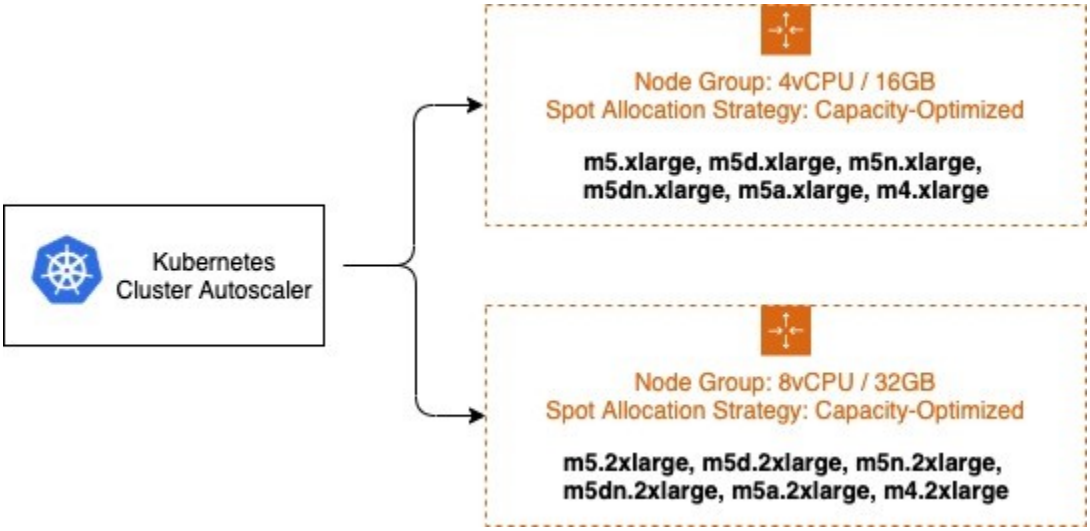
- 每个分片都配置为指向一组唯一的 EC2 Auto Scaling 组
- 每个分片都部署到单独的命名空间，以避免领导者选举冲突

# 针对成本和可用性进行优化

## 竞价型实例

您可以在节点组中使用竞价型实例，比按需价格节省高达 90% 的费用，但要权衡一下，当 EC2 需要恢复容量时，竞价型实例可以随时中断。当您的 EC2 Auto Scaling 组由于可用容量不足而无法扩展时，就会出现容量不足错误。通过选择多个实例系列来最大限度地提高多样性，可以利用许多竞价型容量池来增加实现所需规模的机会，并减少竞价型实例中断对集群可用性的影响。包含 Spot 实例的混合实例策略是在不增加节点组数量的情况下提高多样性的好方法。请记住，如果您需要有保障的资源，请使用 On-Demand 实例而不是竞价型实例。

配置混合实例策略时，所有实例类型都必须具有相似的资源容量，这一点至关重要。自动扩缩程序的调度模拟器使用 InstanceType 中的第一个。 MixedInstancePolicy如果后续的实例类型更大，则在扩展后可能会浪费资源。如果容量较小，您的 Pod 可能会因为容量不足而无法在新实例上进行调度。例如，M4、M5、M5a 和 M5n 实例都具有相似的 CPU 和内存量，是理想的候选实例。MixedInstancePolicyEC2 实例选择器工具可以帮助您识别相似的实例类型。




建议将容量隔离 On-Demand 并竞价到单独的 EC2 Auto Scaling 组中。这比使用基本容量策略更可取，因为调度属性有根本的不同。由于竞价型实例随时会中断（当 EC2 需要恢复容量时），因此用户经常会污染其可抢占节点，这就要求对抢占行为有明确的 Pod 容忍度。这些污点会导致节点的调度属性不同，因此应将它们分成多个 EC2 Auto Scaling 组。

Cluster Autoscaler 有一个扩展器的概念，它为选择要扩展的节点组提供了不同的策略。该策略`--expander=least-waste`是一个不错的通用默认策略，如果您要使用多个节点组来实现竞价型实例的多样化（如上图所述），则可以通过扩展在扩展活动之后最适合利用的组来帮助进一步优化节点组的成本。

## 确定节点组/ASG 的优先级

您也可以使用优先级扩展器配置基于优先级的自动缩放。`--expander=priority`使您的集群能够优先考虑节点组/ASG，如果由于任何原因无法扩展，它将选择优先级列表中的下一个节点组。例如，当您想使用 P3 实例类型时，这很有用，因为它们的 GPU 可以为您的工作负载提供最佳性能，但作为第二个选择，您也可以使用 P2 实例类型。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*p2-node-group.*
    50:
```

```
      - .*p3-node-group.*
```

Cluster Autoscaler 将尝试扩展名为 p3-node-group 的 EC2 Auto Scaling 组。如果此操作在内未成功`--max-node-provision-time`，它将尝试扩展名称为 p2- node-group 的 EC2 Auto Scaling 组。该值默认为 15 分钟，可以减少该值以获得更快的节点组选择，但如果该值太低，可能会导致不必要的扩展。

## 超额配置

Cluster Autoscaler 可确保仅在需要时向集群添加节点，并在未使用时移除节点，从而最大限度地降低成本。这会显著影响部署延迟，因为许多 Pod 将被迫等待节点向上扩展后才能进行调度。节点可能需要几分钟才能使用，这可能会将 Pod 调度延迟增加一个数量级。

此问题可以通过超额配置来缓解，超额配置是以成本换取调度延迟。过度配置是使用优先级为负的临时 pod 实现的，这些容器会占用集群中的空间。当新创建的 pod 不可调度且优先级更高时，临时 pod 将被抢占以腾出空间。然后，临时 Pod 变得不可调度，从而触发集群自动扩缩程序扩展新的超额配置节点。

过度配置还有其他不太明显的好处。在没有过度配置的情况下，高利用率集群的副作用之一是，Pod 将使用 Pod 或 Node Affinity `preferredDuringSchedulingIgnoredDuringExecution` 规则做出不太理想的调度决策。一个常见的用例是使用将高可用性应用程序的 pod 分隔为多个可用区 AntiAffinity。过度配置会显著增加正确区域的节点可用的几率。

对于您的组织来说，过度配置的容量是一个谨慎的业务决策。从本质上讲，它是在性能和成本之间进行权衡。做出此决定的一种方法是确定您的平均扩容频率，然后将其除以向上扩展新节点所需的时间。例如，如果您平均每 30 秒需要一个新节点，而 EC2 需要 30 秒来配置一个新节点，那么单个超额配置节点将确保始终有一个额外的节点可用，从而将调度延迟缩短 30 秒，代价是增加一个 EC2 实例。要改进区域调度决策，请超额配置与 EC2 Auto Scaling 组中可用区数量相等的节点，以确保调度器可以为传入的 Pod 选择最佳区域。

## 防止缩小规模驱逐

移出某些工作负载成本非常高昂。大数据分析、机器学习任务和测试运行器最终将完成，但如果中断，则必须重新启动。Cluster Autoscaler 将尝试缩减任何低于缩小利用率阈值的节点，这将中断该节点上所有剩余的 Pod。通过确保驱逐成本高昂的 Pod 受到集群自动扩缩器识别的标签的保护，可以防止这种情况。

请确保：

- 有注释驱逐吊舱的代价很高 `cluster-autoscaler.kubernetes.io/safe-to-evict=false`

# 高级使用案例

## EBS 卷

持久存储对于构建有状态的应用程序（例如数据库或分布式缓存）至关重要。EBS 卷在 Kubernetes 上启用此用例，但仅限于特定区域。如果在多个可用区之间使用单独的 EBS 卷对每个可用区进行分片，则这些应用程序可以具有高可用性。然后，集群自动扩缩器可以平衡 EC2 自动扩缩组的扩展。

请确保：

- 通过设置 `balance-similar-node-groups=true` 启用节点组平衡。
- 节点组配置的设置完全相同，但可用区和 EBS 卷不同。

## Co-Scheduling

Machine Learning 分布式培训作业可从同区节点配置的最小化延迟中获益匪浅。这些工作负载将多个 Pod 部署到特定区域。这可以通过为所有共同调度的 pod 设置 Pod Affinity 或使用的 Node Affinity 来实现。`topologyKey: failure-domain.beta.kubernetes.io/zone`然后，集群自动扩缩器将扩展特定区域以满足需求。您可能希望分配多个 EC2 Auto Scaling 组，每个可用区一个，以便为整个联合计划的工作负载启用故障转移。

请确保：

- 通过设置 `balance-similar-node-groups=false` 启用节点组平衡
- 当集群同时包含区域和区域@@ 节点组时，将使用 N and/or ode Affinity Pod 抢占。
  - 使用 Node Affinity 强制或鼓励区域 pod 避开区域节点组，反之亦然。
  - 如果区域 Pod 调度到区域节点组，则会导致您的区域 Pod 的容量不平衡。
  - 如果您的区域工作负载可以容忍中断和重新定位，请配置 Pod Preemption 以启用按区域扩展的 pod，以便在竞争较少的区域强制抢占和重新调度。

## 加速器

一些集群利用专门的硬件加速器，例如 GPU。横向扩展时，加速器设备插件可能需要几分钟才能将资源通告到集群。Cluster Autoscaler 模拟该节点将配备加速器，但是在加速器准备就绪并更新节点的可用资源之前，无法在该节点上调度待处理的 Pod。这可能会导致重复、不必要的横向扩展。

此外，即使未使用加速器，CPU 或内存利用率较高的节点也不会考虑缩小规模。由于加速器的相对成本，这种行为可能会很昂贵。相反，如果节点有未占用的加速器，集群自动扩缩程序可以应用特殊规则来考虑缩小规模。

为了确保在这些情况下行为正确，您可以在加速器节点上配置 kubelet，使其在节点加入集群之前对其进行标记。Cluster Autoscaler 将使用此标签选择器来触发加速器优化行为。

请确保：

- 适用于 GPU 节点的 Kubelet 配置为 `--node-labels k8s.amazonaws.com/accelerator=$ACCELERATOR_TYPE`
- 带有加速器的节点遵守上述相同的调度属性规则。

### 从 0 开始缩放

Cluster Autoscaler 能够将节点组缩放到零或从零扩展，这可以节省大量成本。它通过检查 LaunchConfiguration 或 LaunchTemplate中InstanceType 指定的来检测 Auto Scaling 组的 CPU、内存和 GPU 资源。有些 pod 需要额外的资源，例如`WindowsENI`或`PrivateIPv4Address`或特定的 NodeSelectors 或 Taints，这些资源无法从中 LaunchConfiguration发现。集群自动扩缩程序可以通过从 EC2 Auto Scaling 组的标签中发现这些因素来解释这些因素。例如：

```
Key: k8s.io/cluster-autoscaler/node-template/resources/$RESOURCE_NAME
Value: 5
Key: k8s.io/cluster-autoscaler/node-template/label/$LABEL_KEY
Value: $LABEL_VALUE
Key: k8s.io/cluster-autoscaler/node-template/taint/$TAINT_KEY
Value: NoSchedule
```

> **Note**
>
> 请记住，当扩展到零时，您的容量将返回到 EC2，并且将来可能不可用。

## 其他参数

有许多配置选项可用于调整 Cluster Autoscaler 的行为和性能。有关参数的完整列表，请访问GitHub。

| 参数 | 说明 | 默认 |
| --- | --- | --- |
| 扫描间隔 | 为向上或向下扩展而重新评估集群的频率 | 10 秒 |
| 最大空批量删除 | 可以同时删除的最大空节点数。 | 10 |
| 添加后缩小延迟 | 在扩大规模后多久才会恢复缩小规模的评估 | 10 分钟 |
| 删除后缩小延迟 | 节点删除后多长时间恢复缩小评估，默认为扫描间隔 | 扫描间隔 |
| 故障后缩小延迟 | 缩减规模失败后多久可以恢复缩小规模的评估 | 3 分钟 |
| 缩小不需要的时间 | 节点需要多长时间才有资格缩小规模 | 10 分钟 |
| 缩小未准备就绪时间 | 未就绪的节点需要多长时间才有资格缩小规模 | 20 分钟 |
| 缩小利用率阈值 | 节点利用率级别，定义为请求的资源总和除以容量，低于该水平可以考虑缩减节点 | 0.5 |
| 缩小非空候选人数量 | 在一次迭代中考虑作为排水向下缩减的候选节点的最大非空节点数。值越低意味着 CA 响应速度越好，但缩小延迟可能会变慢。较高的值可能会影响大型集群（数百个节点）的 CA 性能。设置为非正值可关闭此启发式算法-CA 不会限制其考虑的节点数量。” | 30 |
| 缩小候选人池比例 | 当先前迭代中的某些候选节点不再有效时，被视为缩减的额 | 0.1 |

| 参数 | 说明 | 默认 |
| --- | --- | --- |
| | 外非空候选节点的比例。值越低意味着 CA 响应速度越好，但缩小延迟可能会变慢。较高的值可能会影响大型集群（数百个节点）的 CA 性能。设置为 1.0 可关闭此启发式算法-CA 会将所有节点作为其他候选节点。 | |
| 缩小候选人池最小数量 | 当先前迭代中的某些候选节点不再有效时，被视为向下缩减的额外非空候选节点的最小数量。在计算其他候选人的人才库规模时，我们会采用 `max(#nodes * scale-down-candidates-pool-ratio, scale-down-candidates-pool-min-count)` | 50 |

## 其他资源

本页包含集群自动扩缩器演示和演示的列表。如果您想在此处添加演示文稿或演示，请发送拉取请求。

| Presentation/Demo | 主持人 |
| --- | --- |
| Kubernetes 上的自动扩缩和成本优化：从 0 到 100 | Guy Templeton、Skyscanner 和 Jiaxin Shan，Amazon |
| SIG-Autoscaling 深度潜水 | Maciek Pytel & Marcin Wielgus |

## 参考

- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md

- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md
- https://github.com/aws/amazon-ec2-instance-selector
- https://github.com/aws/aws-node-termination-handler

# 可靠性最佳实践

本节提供有关如何使在 EKS 上运行的工作负载具有弹性和高可用性的指导

## 如何使用本指南

本指南适用于想要在 EKS 中开发和运营高可用性和容错服务的开发人员和架构师。为了便于阅读，该指南分为不同的主题领域。每个主题都以简要概述开头，然后列出有关您的 EKS 集群可靠性的建议和最佳实践。

## 简介

EKS 的可靠性最佳实践分为以下主题：

- 应用程序
- 控制层面
- 数据层面

是什么使系统可靠？ 如果系统在一段时间内环境发生变化，但仍能始终如一地运行并满足需求，则可以称之为可靠。为实现这一目标，系统必须检测故障，自动自我修复，并能够根据需求进行扩展。

客户可以使用 Kubernetes 作为基础，可靠地运行关键任务应用程序和服务。但是，除了采用基于容器的应用程序设计原则外，可靠地运行工作负载还需要可靠的基础架构。在 Kubernetes 中，基础设施包括控制平面和数据平面。

EKS 提供了生产级 Kubernetes 控制平面，该控制平面旨在实现高可用性和容错性。

在 EKS 中，AWS 负责 Kubernetes 控制平面的可靠性。EKS 在一个 AWS 区域的三个可用区中运行 Kubernetes 控制平面。它会自动管理 Kubernetes API 服务器和 etcd 集群的可用性和可扩展性。

数据平面可靠性的责任由您、客户和 AWS 共同承担。EKS 为部署 Kubernetes 数据平面提供了四个工作节点选项。

EKS Auto Mod e 是最受管理的选项，它可以处理数据平面的配置、扩展和更新，同时提供托管计算、网络和存储功能。自动模式 AMI 经常发布，集群会自动更新到最新 AMI，以部署 CVE 修复和安全补丁。您可以通过在自动模式上配置中断控制来控制何时发生这种情况 NodePools。

Fargate 通过每个节点运行一个 Pod 来处理数据平面的配置和扩展。第三个选项是托管节点组，用于处理数据平面的配置和更新。最后，自我管理的节点是数据平面中管理最少的选项。你使用的 AWS-managed 数据平面越多，你的责任就越少。

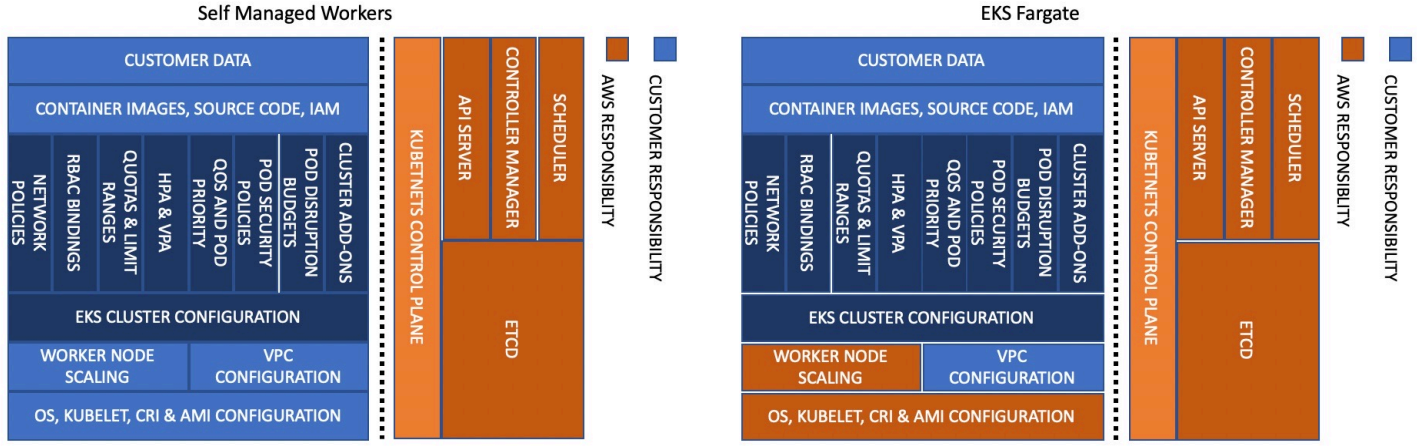
托管节点组可自动执行 EC2 节点的配置和生命周期管理。您可以使用 EKS API（使用 EKS 控制台、AWS API、AWS CLI CloudFormation、Terraform 或`eksctl`）来创建、扩展和升级托管节点。托管节点在您的账户中运行 EKS-optimized Amazon Linux 2 EC2 实例，您可以通过启用 SSH 访问来安装自定义软件包。当您配置托管节点时，它们会作为 EKS-managed Auto Scaling 组的一部分运行，该组可以跨越多个可用区；您可以通过创建托管节点时提供的子网进行控制。EKS 还会自动标记托管节点，以便它们可以与集群自动扩缩器一起使用。

Amazon EKS 在托管节点组上遵循 CVE 和安全补丁的责任共担模式。由于托管节点运行 Amazon EKS-optimized AMI，因此 Amazon EKS 负责在修复错误时构建这些 AMI 的补丁版本。但是，您负责将这些修补的 AMI 版本部署到托管节点组。

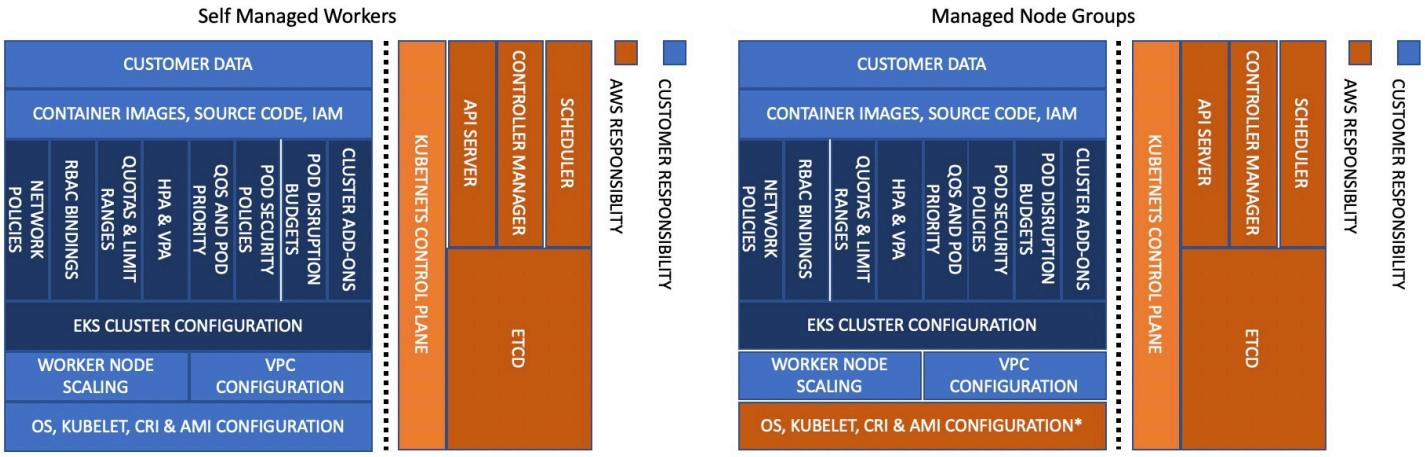
尽管你必须启动更新过程，但EKS也可以管理更新节点。EKS 文档中解释了更新托管节点的过程。

如果您运行自管理节点，则可以使用 Amazon EKS-optimized Linux AMI 创建工作节点。您负责修补和升级 AMI 和节点。最佳做法是使用`eksctl` CloudFormation、或基础架构作为代码工具来配置自我管理节点，因为这样可以轻松升级自我管理的节点。在更新工作节点时，请考虑迁移到新节点，因为迁移过程会将旧的节点组弄成污染，`NoSchedule`并在新堆栈准备好接受现有 Pod 工作负载后耗尽节点。但是，您也可以对自行管理的节点进行就地升级。

责任共担模型——Fargate




责任共担模型-MNG




本指南包括一组建议，您可以使用这些建议来提高 EKS 数据平面、Kubernetes 核心组件和应用程序的可靠性。

# 反馈

本指南即将发布， GitHub 旨在收集来自更广泛 EKS/Kubernetes 社区的直接反馈和建议。如果您认为我们应该在指南中包含最佳实践，请在 GitHub 存储库中提交问题或提交 PR。随着服务中添加新功能或新的最佳实践的发展，我们打算定期更新该指南。

# 运行高可用性应用程序

您的客户希望您的应用程序始终可用，包括在您进行更改时，尤其是在流量激增期间。可扩展且具有弹性的架构可让您的应用程序和服务不受干扰地运行，从而让您的用户满意。可扩展的基础架构会根据业务需求进行增长和缩小。消除单点故障是提高应用程序可用性并使其具有弹性的关键一步。

借助 Kubernetes，您可以以高度可用和弹性的方式操作和运行应用程序。它的声明式管理可确保在设置应用程序后，Kubernetes 会不断尝试将当前状态与所需状态相匹配。

## 建议

### 配置 Pod 中断预算

Pod 中断预算用于限制应用程序将经历的并发中断量。如果必须始终有一部分工作负载可用，则应针对工作负载对其进行配置。EKS 自动模式、Karpenter 和 Cluster Autoscaler 在缩小规模时会知道并遵守配置的 Pod 中断预算。EKS 自动模式、Karpenter 和托管节点组在更新节点时也要遵守吊舱中断预算

## 避免运行单例 Pod

如果你的整个应用程序在单个 Pod 中运行，那么如果该 Pod 被终止，你的应用程序将不可用。与其使用单个 Pod 部署应用程序，不如创建部署。如果 Deployment 创建的 Pod 失败或被终止，部署控制器将启动一个新的 Pod，以确保指定数量的副本 Pod 始终处于运行状态。

## 运行多个副本

使用 Deployment 运行应用程序的多个副本 Pod 有助于其以高度可用的方式运行。如果一个副本失败，剩余的副本仍然可以运行，尽管容量会降低，直到 Kubernetes 创建另一个 Pod 来弥补损失。此外，您可以使用 Horistant Pod Autoscaler 根据工作负载需求自动扩展副本。

## 跨节点调度副本

如果所有副本都在同一个节点上运行，并且该节点变得不可用，那么运行多个副本就没有多大用处。考虑使用 pod 反亲和性或 pod 拓扑分布约束将部署的副本分布到多个工作节点上。

您可以通过跨多个可用区运行典型应用程序来进一步提高其可靠性。

使用 Pod 反关联性规则

下面的清单告诉 Kubernetes 调度器更喜欢将 Pod 放在单独的节点和可用区上。它不需要不同的节点或可用区，因为如果需要不同的节点或可用区，那么一旦每个可用区中都有一个容器在运行，Kubernetes 将无法调度任何 Pod。如果您的应用程序只需要三个副本，则可以使用 `requiredDuringSchedulingIgnoredDuringExecution` for，而 Kubernetes 调度器不会在同一个可用区中调度两个 Pod。`topologyKey: topology.kubernetes.io/zone`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
```

```
        app: web-server
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: topology.kubernetes.io/zone
            weight: 100
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: kubernetes.io/hostname
            weight: 99
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

使用 Pod 拓扑分布约束

与 Pod 反关联性规则类似，Pod 拓扑分布约束允许您在不同的故障（或拓扑）域（例如主机或可用区）中使用您的应用程序。当您尝试通过在每个不同的拓扑域中拥有多个副本来确保容错能力和可用性时，这种方法非常有效。另一方面，Pod 反关联性规则可以很容易地生成拓扑域中只有一个副本的结果，因为彼此具有反亲和力的 Pod 会产生排斥效果。在这种情况下，专用节点上的单个副本不适合容错，也不能很好地利用资源。使用拓扑分布约束，您可以更好地控制调度器应尝试在拓扑域中应用的分布或分布。以下是这种方法中要使用的一些重要属性：

1. maxSkew用于控制或确定拓扑域中事物不均匀的最大点。例如，如果一个应用程序有 10 个副本并且部署在 3 个可用区，则无法实现均匀分布，但您可以影响分布的不均衡程度。在这种情况下，maxSkew可以是 1 到 10 之间的任何值。值为 1 意味着您最终可能会得到类似于 3 个可用区的点差4,3,3，3,4,3或者3,3,4跨过 3 个可用区。相比之下，值为 10 意味着您最终可能会获得类似10,0,00,10,0或0,0,10跨越 3 个可用区的点差。

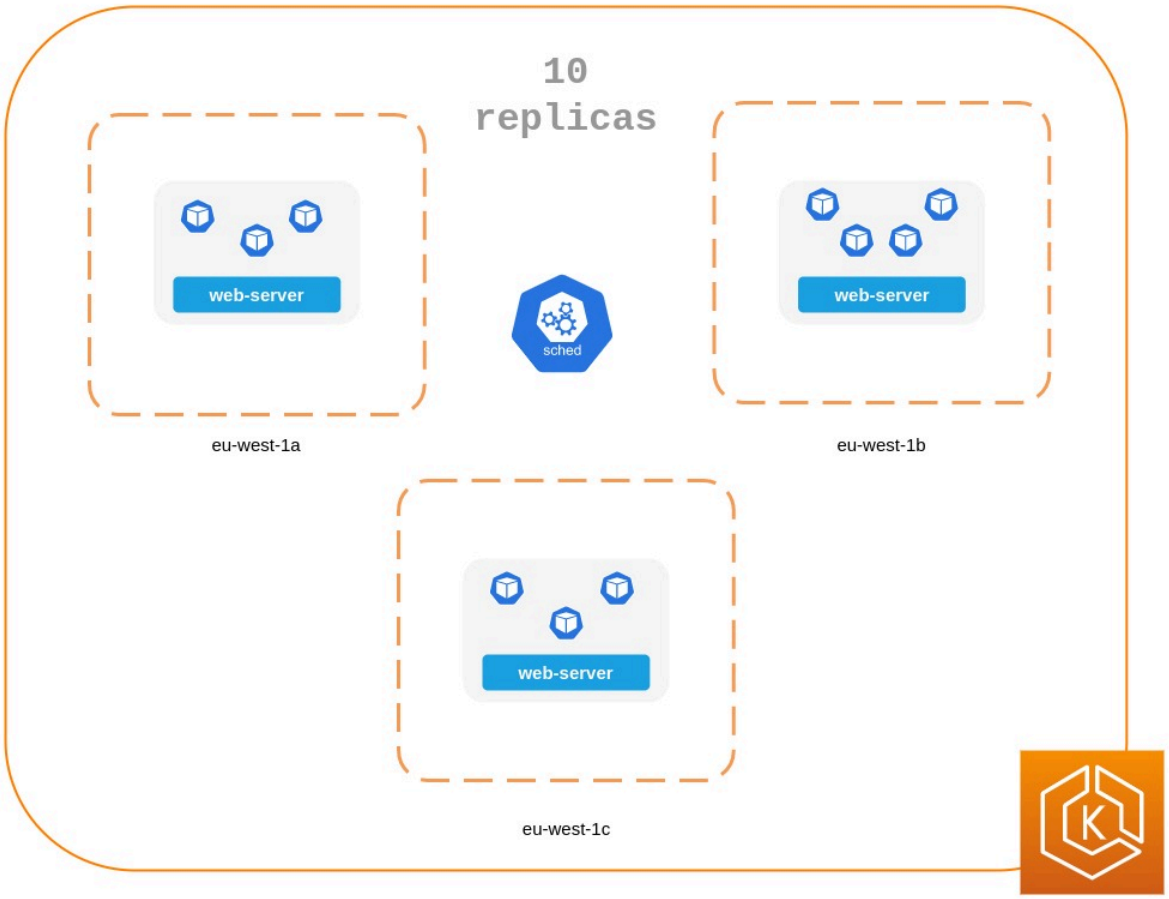
2. topologyKey是其中一个节点标签的密钥，它定义了应用于 Pod 分发的拓扑域的类型。例如，区域点差将具有以下键值对：

```
topologyKey: "topology.kubernetes.io/zone"
```

3. 该whenUnsatisfiable属性用于确定在无法满足所需约束条件时您希望调度器如何响应。
4. 用于查找匹配labelSelector的 pod，以便调度器在根据您指定的约束条件决定将 Pod 放置在何处时可以意识到它们。

除了上述内容之外，您还可以在 Kubernetes 文档中进一步阅读其他字段。

Pod 拓扑在 3 个可用区之间分布限制




```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 10
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

### 运行 Kubernetes 指标服务器

安装 Kubernetes 指标服务器以帮助扩展您的应用程序。像 HPA 和 VPA 这样的 Kubernetes 自动扩缩程序插件需要跟踪应用程序的指标才能对其进行扩展。指标服务器收集可用于做出扩展决策的资源指标。这些指标是从 kubelets 收集的，并以指标 API 格式提供。

指标服务器不保留任何数据，也不是监控解决方案。其目的是向其他系统公开 CPU 和内存使用率指标。如果您想跟踪应用程序在一段时间内的状态，则需要像 Prometheus 或 Amazon 这样的监控工具。 CloudWatch

按照 EKS 文档在您的 EKS 集群中安装指标服务器。

## 水平吊舱自动扩缩器 (HPA)

HPA 可以根据需求自动扩展您的应用程序，并帮助您避免在流量高峰期影响客户。它在 Kubernetes 中实现为控制循环，定期从提供资源指标的 API 中查询指标。

HPA 可以从以下 API 检索指标：1. `metrics.k8s.io`也称为资源指标 API — 提供 pod 2 的 CPU 和内存使用情况。 `custom.metrics.k8s.io`— 提供来自其他指标收集器（例如 Prometheus）的指标；这些指标是你的 Kubernetes 集群内部的。3。 `external.metrics.k8s.io`— 提供 Kubernetes 集群外部的指标（E.g.、SQS 队列深度、弹性负载均衡延迟）。

您必须使用这三个 API 中的一个来提供扩展应用程序的指标。

### 根据自定义或外部指标扩展应用程序

您可以使用自定义或外部指标，根据 CPU 或内存利用率以外的指标来扩展应用程序。自定义指标 API 服务器提供了 HPA 可用于自动扩展`custom-metrics.k8s.io`应用程序的 API。

你可以使用 Prometheus Adapter for Kubernetes 指标 API 从 Prometheus 收集指标并与 HPA 一起使用。在这种情况下，Prometheus 适配器将以 Metrics API 格式公开 Prometheus 指标。

部署 Prometheus 适配器后，即可使用 kubectl 查询自定义指标。 `kubectl get –raw /apis/custom.metrics.k8s.io/v1beta1/`

顾名思义，外部指标使横向 Pod Autoscaler 能够使用 Kubernetes 集群外部的指标来扩展部署。例如，在批处理工作负载中，通常会根据 SQS 队列中正在运行的作业数量来扩展副本的数量。

要自动扩展 Kubernetes 工作负载，你可以使用 KEDA（Kubernetes Event-driven Autoscaling），这是一个开源项目，可以根据许多自定义事件推动容器扩展。这篇 AWS 博客概述了如何使用适用于 Prometheus 的亚马逊托管服务来实现 Kubernetes 工作负载自动扩展。

## 垂直吊舱自动扩缩器 (VPA)

VPA 会自动调整 Pod 的 CPU 和内存预留，以帮助您“调整应用程序的大小”。对于需要垂直扩展的应用程序（通过增加资源分配来完成），您可以使用 VPA 自动扩展 Pod 副本或提供扩展建议。

如果 VPA 需要对其进行扩展，您的应用程序可能会暂时不可用，因为 VPA 的当前实现不会对 Pod 进行就地调整；相反，它会重新创建需要扩展的 Pod。

EKS 文档包括设置 VPA 的演练。

Fairwinds Goldilocks 项目提供了一个仪表板，用于可视化针对CPU和内存请求和限制的VPA建议。它的 VPA 更新模式允许你根据 VPA 的建议自动缩放 Pod。

## 更新应用程序

现代应用程序需要快速创新，并具有高度的稳定性和可用性。Kubernetes 为你提供了在不中断客户的情况下持续更新应用程序的工具。

让我们来看看一些能够在不牺牲可用性的情况下快速部署变更的最佳实践。

## 有执行回滚的机制

拥有撤消按钮可以避免灾难。在更新生产集群之前，最佳做法是在单独的较低环境（测试或开发环境）中测试部署。使用 CI/CD管道可以帮助您自动化和测试部署。通过持续部署管道，如果升级碰巧有缺陷，您可以快速恢复到旧版本。

您可以使用部署来更新正在运行的应用程序。这通常是通过更新容器镜像来完成的。你可以用它kubectl来更新部署，如下所示：

```
kubectl --record deployment.apps/nginx-deployment set image nginx-deployment
 nginx=nginx:1.16.1
```

该--record参数记录了对部署的更改，并在需要执行回滚时为您提供帮助。 kubectl rollout history deployment显示了集群中已记录的部署更改。您可以使用回滚更改。kubectl rollout undo deployment <DEPLOYMENT_NAME>

默认情况下，当你更新需要重新创建 pod 的部署时，Deployment 将执行滚动更新。换句话说，Kubernetes 只会更新部署中正在运行的 Pod 的一部分，而不是一次更新所有 Pod。你可以通过属性控制 Kubernetes 如何执行滚动更新。RollingUpdateStrategy

在对 Deployment 执行滚动更新时，您可以使用Max Unavailable属性来指定更新期间不可用的 Pod 的最大数量。Deployment 的Max Surge属性允许你设置在所需的 Pod 数量之上可以创建的最大 Pod 数量。

考虑max unavailable进行调整，以确保推出不会干扰您的客户。例如，Kubernetes max unavailable 默认设置为 25%，这意味着如果你有 100 个 Pod，那么在推出期间可能只有 75 个 Pod 处于活动状态。如果您的应用程序至少需要 80 个 Pod，则此部署可能会造成中断。相反，您可以max unavailable将设置为 20%，以确保在整个部署过程中至少有 80 个功能性的 Pod。

## 使用 blue/green 部署

变更本质上是有风险的，但是无法撤消的更改可能会造成灾难性的后果。更改程序允许您通过回滚有效地将时间倒流，从而使增强功能和实验更加安全。 Blue/green 部署为您提供了一种在出现问题时快速撤回更改的方法。在此部署策略中，您将为新版本创建环境。此环境与正在更新的应用程序的当前版本相同。配置新环境后，流量就会路由到新环境。如果新版本在不产生错误的情况下产生预期的结果，则旧环境将被终止。否则，流量将恢复到旧版本。

您可以通过创建与现有版本 blue/green 部署相同的新部署来在 Kubernetes 中执行部署。确认新部署中的 Pod 运行没有错误后，您可以通过更改服务中将流量路由到应用程序的 Pod 的`selector`规范，开始向新部署发送流量。

许多持续集成工具，例如 Flux、Jenkins 和 Spinn aker，都允许你实现部署的自动化。 blue/green AWS 容器博客包括使用 AWS Load Balancer 控制器的演练：使用 AWS Load Balancer 控制器进行 blue/green 部署、金丝雀部署和测试 A/B

### 使用 Canary 部署

Canary 部署是 blue/green 部署的一种变体，可以显著消除变更带来的风险。在此部署策略中，除了旧部署之外，您还创建了一个包含更少 Pod 的新部署，并将一小部分流量转移到新的部署。如果指标表明新版本的性能与现有版本一样好或更好，则可以逐步增加新部署的流量，同时扩大其规模，直到所有流量都转移到新部署。如果出现问题，您可以将所有流量路由到旧部署，并停止向新部署发送流量。

尽管 Kubernetes 没有提供原生方式来执行金丝雀部署，但你可以在 Istio 中使用 Fl ag ger 等工具。

## Health 检查和自我修复

没有哪个软件是没有错误的，但是 Kubernetes 可以帮助你最大限度地减少软件故障的影响。过去，如果应用程序崩溃，则必须有人通过手动重启应用程序来纠正这种情况。Kubernetes 使您能够检测 Pod 中的软件故障，并自动将其替换为新的副本。借助 Kubernetes，您可以监控应用程序的运行状况并自动替换运行状况不佳的实例。

Kubernetes 支持三种类型的运行状况检查：

1. 活性探测器
2. 启动探测（在 Kubernetes 1.16+ 版本中支持）
3. 准备情况调查

Kubernetes 代理 Kubel et 负责运行上述所有检查。Kubelet 可以通过三种方式检查 Pod 的生命值：kubelet 可以在容器内运行 shell 命令，向其容器发送 HTTP GET 请求，或者在指定端口上打开 TCP 套接字。

如果您选择`exec`基于探测器，在容器内运行 shell 脚本，请确保 shell 命令在`timeoutSeconds`值到期之前退出。否则，您的节点将有`<defunct>`进程，从而导致节点故障。

# 建议

## 使用 Liveness Probe 移除不健康的 pod

Liveness 探测器可以检测到进程继续运行但应用程序无响应的死锁情况。例如，如果您正在运行在端口 80 上侦听的 Web 服务，则可以将 Liveness 探测器配置为在 Pod 的端口 80 上发送 HTTP GET 请求。Kubelet 会定期向 Pod 发送 GET 请求并期望得到响应；如果 Pod 在 200-399 之间做出响应，那么 kubelet 会认为 Pod 运行状况良好；否则，Pod 将被标记为运行状况不佳。如果 Pod 持续未通过运行状况检查，kubelet 会将其终止。

您可以使用`initialDelaySeconds`来延迟第一次探测。

使用 Liveness Probe 时，请确保您的应用程序不会遇到所有 Pod 同时无法通过 Liveness 探测的情况，因为 Kubernetes 会尝试替换所有 Pod，这将使您的应用程序离线。此外，Kubernetes 将继续创建新的 Pod，这些容器也将无法通过 Liveness Probes，这给控制平面带来不必要的压力。避免将 Liveness Probe 配置为依赖于 Pod 外部的因素，例如外部数据库。换句话说，对你的 Pod 外部数据库没有响应不应使你的 Pod 无法通过 Liveness Probes。

Sandor Szücs 的帖子 LIVENESS PROBES ARE DANGERS 描述了探测器配置错误可能导致的问题。

## 对于需要更长时间才能启动的应用程序，请使用 Startup Probe

当您的应用程序需要更多时间才能启动时，您可以使用启动探测器来延迟存活和就绪探测器。例如，需要整合数据库缓存的 Java 应用程序可能需要长达两分钟的时间才能完全运行。在完全运行之前，任何存活或就绪探测都可能失败。配置启动探测将允许 Java 应用程序在执行 Liveness 或 Readiness Probe 之前恢复正常。

在启动探测成功之前，所有其他探测器都将被禁用。你可以定义 Kubernetes 等待应用程序启动的最长时间。如果在最大配置时间之后，Pod 仍然无法通过启动探测，则它将被终止，并创建一个新的 Pod。

启动探测器与 Liveness Probe 类似，如果它们失败，则会重新创建 Pod。正如 Ricardo A. 在他的《神奇探测器及其配置方法》一文中所解释的那样，当应用程序的启动时间不可预测时，应使用启动探测器。如果您知道应用程序需要十秒钟才能启动，则应`initialDelaySeconds`改用 Liveness/Readiness Probe 和。

## 使用就绪探测器检测部分不可用

当 Liveness 探测器在应用程序中检测到通过终止 Pod（因此重新启动应用程序）来解决的故障时，Readiness Probe 会检测应用程序可能暂时不可用的情况。在这些情况下，应用程序可能会暂时失去响应；但是，该操作完成后，预计它会恢复正常。

例如，在密集的磁盘 I/O 操作期间，应用程序可能暂时无法处理请求。在这里，终止应用程序的 Pod 不是一种补救措施；同时，发送到 Pod 的其他请求可能会失败。

你可以使用 Readiness Probe 来检测应用程序中的暂时不可用，并停止向其 Pod 发送请求，直到它恢复正常运行。与 Liveness Probe 不同，失败会导致 Pod 的重新创建，而失败的就绪探测将意味着 Pod 不会收到来自 Kubernetes 服务的任何流量。当就绪探测成功后，Pod 将恢复接收来自服务的流量。

就像 Liveness Probe 一样，避免配置依赖于 Pod 外部资源（例如数据库）的就绪探针。以下是配置不当的 Readiness 会导致应用程序无法运行的场景——如果 Pod 的就绪探测在应用程序的数据库无法访问时失败，那么其他 Pod 副本也会同时失败，因为它们具有相同的运行状况检查标准。以这种方式设置探测器可以确保每当数据库不可用时，Pod 的就绪探测器就会失败，并且 Kubernetes 将停止向所有 Pod 发送流量。

使用就绪探测器的副作用是，它们可能会增加更新部署所需的时间。除非就绪探测成功，否则新副本将不会接收流量；在此之前，旧副本将继续接收流量。

## 应对中断

Pod 的寿命是有限的，即使你有长时间运行的 Pod，也要谨慎地确保 Pod 在时机成熟时正确终止。根据您的升级策略，Kubernetes 集群升级可能需要您创建新的工作节点，这需要在较新的节点上重新创建所有 Pod。适当的终止处理和 Pod 中断预算可以帮助您避免服务中断，因为 Pod 会从较旧的节点中移除，并在较新的节点上重新创建。

升级工作节点的首选方法是创建新的工作节点并终止旧的工作节点。在终止工作节点之前，你应该`drain`这样做。当工作节点耗尽时，其所有 pod 都会被安全驱逐出去。安全是这里的关键词；当工作人员身上的吊舱被驱逐时，它们不会简单地收到`SIGKILL`信号。取而代之的是，向被驱逐的 Pod 中每个容器的主进程 (PID 1) 发送`SIGTERM`信号。`SIGTERM`信号发送后，Kubernetes 将在发送`SIGKILL`信号之前给进程一段时间（宽限期）。默认情况下，此宽限期为 30 秒；您可以通过在 kubectl 中使用`grace-period`标志或在 Podspec `terminationGracePeriodSeconds` 中声明来覆盖默认值。

```
kubectl delete pod <pod name> –grace-period=<seconds>
```

通常会有主进程没有 PID 1 的容器。考虑这个 Python-based 示例容器：

```
$ kubectl exec python-app -it ps
 PID USER TIME COMMAND
 1   root 0:00 {script.sh} /bin/sh ./script.sh
 5   root 0:00 python app.py
```

在此示例中，shell 脚本接收SIGTERM，但主进程（在本示例中恰好是 Python 应用程序）没有收到SIGTERM信号。当 Pod 被终止时，Python 应用程序将被突然终止。这可以通过更改容器来启动 Python 应用程序来修复。ENTRYPOINT或者，你可以使用像 dumb-init 这样的工具来确保你的应用程序可以处理信号。

您还可以使用容器挂钩在容器启动或停止时执行脚本或 HTTP 请求。PreStop挂钩操作在容器收到SIGTERM信号之前运行，并且必须在发送此信号之前完成。该terminationGracePeriodSeconds值从PreStop挂钩操作开始执行时起生效，而不是SIGTERM信号发送时生效。

## 建议

### 使用 Pod 中断预算保护关键工作负载

如果应用程序的副本数量低于声明的阈值，Pod 中断预算或 PDB 可以暂时停止驱逐过程。一旦可用副本数量超过阈值，驱逐过程将继续进行。您可以使用 PDB 来声明副本的maxUnavailable数量minAvailable和数量。例如，如果您希望应用程序至少有三个副本可用，则可以创建一个 PDB。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: my-svc-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: my-svc
```

上述 PDB 策略告诉 Kubernetes 在有三个或更多副本可用之前停止驱逐过程。节点耗尽尊重PodDisruptionBudgets。在 EKS 托管节点组升级期间，节点会被耗尽，超时时间为十五分钟。十五分钟后，如果未强制更新（该选项在 EKS 控制台中称为“滚动更新”），则更新将失败。如果强制更新，则会删除 pod。

对于自我管理的节点，您还可以使用诸如 AWS 节点终止处理程序之类的工具，该工具可确保 Kubernetes 控制平面对可能导致您的 EC2 实例不可用的事件（例如 EC2 维护事件和 EC 2 Spot 中断）做出适当的响应。它使用 Kubernetes API 封锁节点以确保没有新的 Pod 被调度，然后将其耗尽，终止所有正在运行的 Pod。

你可以使用 Pod 反亲和性在不同的节点上调度 Deployment 的 Pod，避免节点升级期间与 PDB 相关的延迟。

### 练习混沌工程

Chaos Engineering 是一门在分布式系统上进行实验的学科，目的是建立人们对系统承受生产中动荡条件的能力的信心。

多米尼克·托尔诺（Dominik Tor now）在他的博客中解释说，Kubernetes是一个声明式系统，其中“用户向系统提供所需系统状态的表示形式。然后，系统会考虑当前状态和所需状态，以确定从当前状态过渡到所需状态的命令顺序。“ 这意味着 Kubernetes 始终存储所需的状态，如果系统出现偏差，Kubernetes 将采取措施恢复状态。例如，如果工作节点不可用，Kubernetes 会将这些 Pod 重新调度到另一个工作节点上。同样，如果`replica`崩溃，部署控制器将创建一个新的。`replica`通过这种方式，Kubernetes 控制器可以自动修复故障。

像 Gremlin 这样的混沌工程工具可以帮助你测试 Kubernetes 集群的弹性并识别单点故障。在集群（及其他集群）中引入人为混乱的工具可以发现系统性弱点，为识别瓶颈和错误配置提供机会，并在受控环境中纠正问题。Chaos Engineering 的理念主张故意破坏事物，并对基础设施进行压力测试，以最大限度地减少意想不到的停

### 使用服务网格

您可以使用服务网格来提高应用程序的弹性。服务网格支持服务间的通信并提高微服务网络的可观察性。大多数服务网格产品的工作原理是在每个服务旁边运行一个小型网络代理，用于拦截和检查应用程序的网络流量。您可以将应用程序放置在网格中，而无需修改应用程序。使用服务代理的内置功能，您可以让它生成网络统计信息、创建访问日志，以及向出站请求添加 HTTP 标头以进行分布式跟踪。

服务网格可通过自动请求重试、超时、断路和速率限制等功能，帮助您提高微服务的弹性。

如果您运营多个集群，则可以使用服务网格来实现跨集群的服务间通信。

### 服务网格

- Istio
- LinkerD
- 领事

## 可观测性

可观察性是一个总称，包括监控、记录和跟踪。基于微服务的应用程序本质上是分布式的。与仅仅监控单个系统的单片应用程序不同，在分布式应用程序架构中，您需要监控每个组件的性能。您可以使用集群级别的监控、日志记录和分布式跟踪系统来识别集群中的问题，以免它们对客户造成干扰。

Kubernetes 内置的故障排除和监控工具有限。指标服务器收集资源指标并将其存储在内存中，但不会将其保存。你可以使用 kubectl 查看 Pod 的日志，但是 Kubernetes 不会自动保留日志。分布式跟踪的实现要么在应用程序代码级别完成，要么使用服务网格完成。

Kubernetes 的可扩展性在这里大放异彩。Kubernetes 允许您使用首选的集中式监控、日志和跟踪解决方案。

## 建议

### 监控您的应用程序

在现代应用程序中，您需要监控的指标数量在不断增长。如果你有一种自动化的方式来跟踪应用程序，这样你就可以专注于解决客户的难题，这会有所帮助。 Cluster-wide 诸如 Prometheus CloudWatch 或 Container Insights 之类的监控工具可以监控您的集群和工作负载，并在出现问题时或最好在出现问题之前向您提供信号。

监控工具允许您创建运营团队可以订阅的警报。考虑使用规则来激活警报，这些事件如果加剧可能会导致中断或影响应用程序性能。

如果您不清楚应该监控哪些指标，可以从以下方法中汲取灵感：

- 红色方法。代表请求、错误和持续时间。
- 使用方法。代表利用率、饱和度和错误。

Sysdig 在 Kubernetes 上发布警报的最佳实践一文中列出了可能影响应用程序可用性的组件。

### 使用 Prometheus 客户端库公开应用程序指标

除了监控应用程序状态和汇总标准指标外，您还可以使用 Prometheus 客户端库公开特定于应用程序的自定义指标，以提高应用程序的可观察性。

### 使用集中式日志工具收集和保留日志

登录 EKS 分为两类：控制平面日志和应用程序日志。EKS 控制平面日志记录可直接从控制平面向账户中的 CloudWatch 日志提供审计和诊断日志。应用程序日志是由集群内运行的 Pod 生成的日志。应用程序日志包括运行业务逻辑应用程序的 Pod 和 Kubernetes 系统组件（例如 CoreDNS、Cluster Autoscaler、Prometheus 等）生成的日志。

EKS 提供五种类型的控制平面日志：

1. Kubernetes API 服务器组件日志
2. 审核
3. 身份验证器
4. 控制器管理器
5. 调度器

控制器管理器和调度程序日志可以帮助诊断控制平面问题，例如瓶颈和错误。默认情况下，EKS 控制平面日志不会发送到 CloudWatch 日志。您可以启用控制平面日志记录，并选择要为账户中的每个集群捕获的 EKS 控制平面日志类型

收集应用程序日志需要在集群中安装日志聚合器工具，例如 Fluen t Bit、Fluentd 或 Contain CloudWatcher Insigh ts。

Kubernetes 日志聚合器工具以节点身份运行 DaemonSets 并从节点中抓取容器日志。然后，应用程序日志被发送到一个集中的目标进行存储。例如，Conta CloudWatch iner Insights 可以使用 Fluent Bit 或 Fluentd 来收集日志并将其发送到CloudWatch 日志进行存储。Fluent Bit 和 Fluentd 支持许多流行的日志分析系统，例如 Elasticsearch 和 InfluxDB，使您能够通过修改 Fluent bit 或 Fluentd 的日志配置来更改日志的存储后端。

## 使用分布式跟踪系统来识别瓶颈

典型的现代应用程序的组件分布在网络上，其可靠性取决于组成该应用程序的每个组件的正常运行。您可以使用分布式跟踪解决方案来了解请求的流向以及系统的通信方式。跟踪可以向您显示应用程序网络中存在瓶颈的位置，并防止可能导致级联故障的问题。

在应用程序中实现跟踪有两种选择：您可以使用共享库在代码级别实现分布式跟踪，也可以使用服务网格。

在代码级别实现跟踪可能不利。在这种方法中，你必须对你的代码进行更改。如果您有多语言应用程序，则情况会更加复杂。您还负责在服务中维护另一个图书馆。

LinkerD 和 Istio 等服务网格可用于在应用程序中实现分布式跟踪，只需对应用程序代码进行最少的更改。您可以使用服务网格来标准化指标生成、记录和跟踪。

像 AWS X-Ray、Jaeger 这样的追踪工具支持共享库和服务网格实现。

考虑使用像 AWS X-Ray 或 Jaeger 这样同时支持（共享库和服务网格）实现的跟踪工具，这样以后采用服务网格时就不必切换工具了。

# EKS 控制面板

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Amazon Elastic Kubernetes Service (EKS) 是一项托管 Kubernetes 服务，可让您轻松地在 AWS 上运行 Kubernetes，而无需安装、操作和维护自己的 Kubernetes 控制平面或工作节点。它在上游 Kubernetes 上游运行，并已通过 Kubernetes 标准认证。这种一致性可确保 EKS 支持 Kubernetes API，就像您可以在 EC2 上或本地安装的开源社区版本一样。在上游 Kubernetes 上运行的现有应用程序与 Amazon EKS 兼容。

EKS 会自动管理 Kubernetes 控制平面节点的可用性和可扩展性，并自动替换运行状况不佳的控制平面节点。

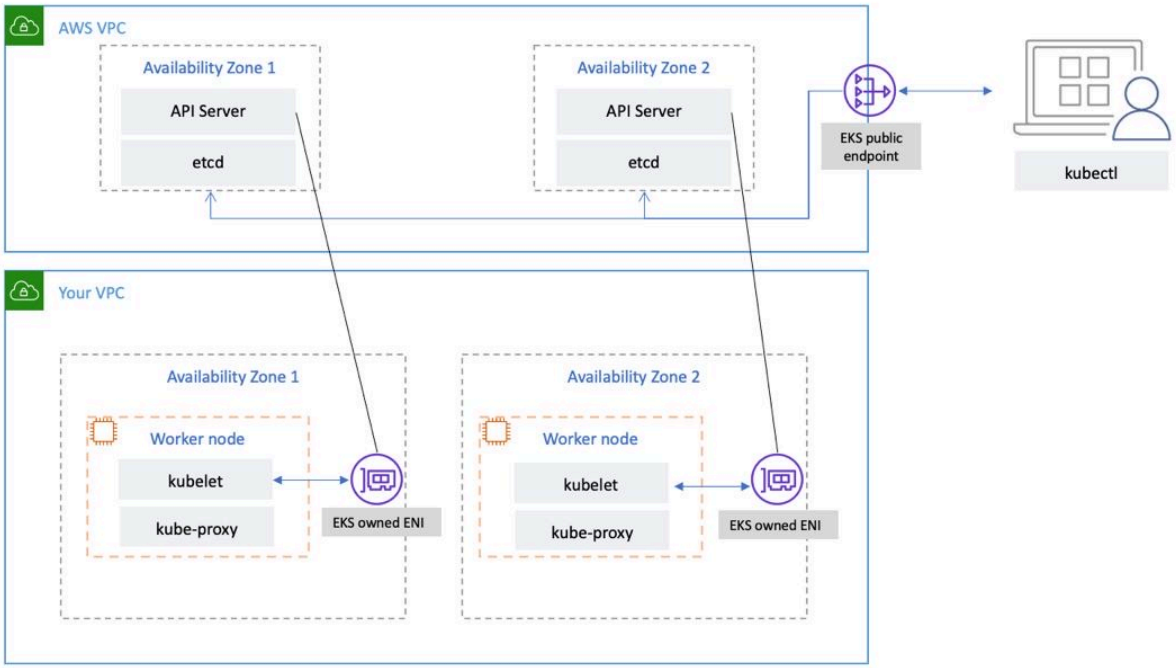
## EKS 架构

EKS 架构旨在消除任何可能影响 Kubernetes 控制平面可用性和耐久性的单点故障。

由 EKS 管理的 Kubernetes 控制平面在 EKS 托管的 VPC 内运行。EKS 控制平面包括 Kubernetes API 服务器节点、etcd 集群。运行 API 服务器、调度器等组件并在自动缩放组`kube-controller-manager`中运行的 Kubernetes API 服务器节点。EKS 在一个 AWS 区域内的不同可用区 (AZ) 中至少运行两个 API 服务器节点。同样，为了持久起见，etcd 服务器节点也在跨越三个可用区的自动缩放组中运行。EKS 在每个可用区运行一个 NAT 网关，API 服务器和 etcd 服务器在私有子网中运行。这种架构可确保单个可用区中的事件不会影响 EKS 集群的可用性。

当您创建新集群时，Amazon EKS 会为托管 Kubernetes API 服务器创建一个高度可用的终端节点，用于与集群通信（使用诸如此类的工具）。`kubectl`托管端点使用 NLB 对 Kubernetes API 服务器进行负载平衡。EKS 还会在不同的可用区中配置两个 ENI，以便于与您的工作节点通信。

EKS 数据平面网络连接




你可以配置你的 Kubernetes 集群的 API 服务器是可以从公共互联网（使用公共终端节点）还是通过你的 VPC（使用 EKS-managed ENI）访问，或者两者兼而有之。

无论用户和工作节点使用公共终端节点还是 EKS-managed ENI 连接到 API 服务器，都有冗余的连接路径。

## 建议

查看以下建议。

## 监控控制平面指标

监控 Kubernetes API 指标可以让你深入了解控制平面性能并发现问题。不健康的控制平面可能会影响集群内运行的工作负载的可用性。例如，写得不好的控制器会使 API 服务器过载，从而影响应用程序的可用性。

Kubernetes 在端点公开了控制平面指标。`/metrics`

您可以使用以下方式查看公开的指标`kubectl`：

```
kubectl get --raw /metrics
```

这些指标以 Prometheus 文本格式表示。

您可以使用 Prometheus 来收集和存储这些指标。2020 年 5 月，在 Container Insights 中CloudWatch 增加了对监控 Prometheus 指标的支持。 CloudWatch因此，您也可以使用 Amazon CloudWatch 来监控 EKS 控制平面。您可以使用添加新 Prometheus 抓取目标教程：Prometheus KPI 服务器指标来收集指标并创建仪表板来监控集群的控制平面。 CloudWatch

你可以在这里找到 Kubernetes API 服务器指标。例
如，`apiserver_request_duration_seconds`可以指示 API 请求需要多长时间才能运行。

考虑监控以下控制平面指标：

## API 服务器

| 指标 | 说明 |
| --- | --- |
| `apiserver_request_total` | 对于每个动词、试运行值、组、版本、资源、范围、组件和 HTTP 响应代码划分的 apiserver 请求的计数器。 |
| `apiserver_request_duration_seconds*` | 每个动词、试运行值、组、版本、资源、子资源、范围和组件的响应延迟直方图（以秒为单位）。 |
| `apiserver_admission_controller_admission_duration_seconds*` | 准入控制器延迟直方图（以秒为单位），按名称标识，并按每个操作以及 API 资源和类型（验证或承认）进行细分。 |
| `apiserver_admission_webhook_rejection_count` | Webhook 被拒绝的入场次数。由名称、操作、拒绝代码、类型（验证或承认）、错误类型（calling_webhook_error、apiserver_internal_error、no_error、no_error、no_error）标识 |
| `rest_client_request_duration_seconds*` | 请求延迟直方图（以秒为单位）。按动词和网址细分。 |
| `rest_client_requests_total` | HTTP 请求数，按状态码、方法和主机分区。 |

- 直方图指标包括 _bucket、_sum 和 _count 后缀。

## etcd

| 指标 | 说明 |
| --- | --- |
| etcd_request_duration_seconds* | 每种操作和对象类型的 Etcd 请求延迟直方图（以秒为单位）。 |
| apiserver_storage_db_total_size_in_bytes 或者apiserver_storage_size_bytes （从 EKS v1.28 开始） | etcd 数据库大小。 |

- 直方图指标包括 _bucket、_sum 和 _count 后缀。

考虑使用 Kubernetes 监控概述控制面板来可视化和监控 Kubernetes API 服务器请求以及延迟和 etcd 延迟指标。

> **Important**
>
> 当超过数据库大小限制时，etcd 会发出空格警报并停止接受进一步的写入请求。换句话说，集群变为只读，所有变异对象（例如创建新 Pod、扩展部署等）的请求都将被集群的 API 服务器拒绝。

## 集群认证

EKS 目前支持两种类型的身份验证：bearer/service账户令牌和使用 webhook 令牌身份验证的 IAM 身份验证。当用户调用 Kubernetes API 时，Webhook 会将请求中包含的身份验证令牌传递给 IAM。该令牌是一个基于 64 的签名 URL，由 AWS 命令行界面 (A WS CLI) 生成。

创建 EKS 集群的 IAM 用户或角色会自动获得对集群的完全访问权限。您可以通过编辑 aws-auth 配置映射来管理对 EKS 集群的访问权限。

如果您错误配置了`aws-auth`配置映射并失去了对集群的访问权限，您仍然可以使用集群创建者的用户或角色来访问您的 EKS 集群。

万一您无法在 AWS 区域使用 IAM 服务，也可以使用 Kubernetes 服务账户的不记名令牌来管理集群。

创建一个允许在集群中执行所有操作的`super-admin`账户：

```
kubectl -n kube-system create serviceaccount super-admin
```

创建赋予超级管理员集群管理员角色的角色绑定：

```
kubectl create clusterrolebinding super-admin-rb --clusterrole=cluster-admin --
serviceaccount=kube-system:super-admin
```

获取服务账号的秘密：

```
SECRET_NAME=`kubectl -n kube-system get serviceaccount/super-admin -o
 jsonpath='{.secrets[0].name}'`
```

获取与密钥关联的代币：

```
TOKEN=`kubectl -n kube-system get secret $SECRET_NAME -o jsonpath='{.data.token}'|
 base64 --decode`
```

将服务账号和令牌添加到kubeconfig：

```
kubectl config set-credentials super-admin --token=$TOKEN
```

将当前上下文设置kubeconfig为使用超级管理员帐户：

```
kubectl config set-context --current --user=super-admin
```

决赛kubeconfig应该是这样的：

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:<REDACTED>
    server: https://<CLUSTER>.gr7.us-west-2.eks.amazonaws.com
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
contexts:
- context:
    cluster: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
    user: super-admin
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
current-context: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
kind: Config
preferences: {}
users:
#- name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
#  user:
#    exec:
#      apiVersion: client.authentication.k8s.io/v1beta1
#      args:
#      - --region
#      - us-west-2
#      - eks
#      - get-token
#      - --cluster-name
#      - <<cluster name>>
```

```
#      command: aws
#      env: null
- name: super-admin
  user:
    token: <<super-admin sa's secret>>
```

## 入学网络挂钩

Kubernetes 有两种类型的准入 webhook：验证准入 webhook 和变更准入 web hook。它们允许用户扩展 kubernetes API，并在对象被 API 接受之前对其进行验证或变异。这些 webhook 的配置不佳可能会阻塞集群的关键操作，从而破坏 EKS 控制平面的稳定。

为了避免影响集群的关键操作，要么避免像下面这样设置 “包罗万象” 的 webhook：

```
- name: "pod-policy.example.com"
  rules:
  - apiGroups:   ["*"]
    apiVersions: ["*"]
    operations:  ["*"]
    resources:   ["*"]
    scope: "*"
```

或者确保 webhook 具有超时时间小于 30 秒的失效开放策略，以确保在您的 webhook 不可用时，它不会损害集群的关键工作负载。

### 使用不安全的 s **y** sctl 封锁 Pod

Sysctl是一个 Linux 实用程序，允许用户在运行时修改内核参数。这些内核参数控制操作系统行为的各个方面，例如网络、文件系统、虚拟内存和进程管理。

Kubernetes 允许为 Pod 分配sysctl配置文件。Kubernetes 将其归类systcls为安全和不安全。Safe sysctls 是容器或 Pod 中的命名空间，设置它们不会影响节点上的其他 Pod 或节点本身。相比之下，不安全的 sysctl 在默认情况下处于禁用状态，因为它们可能会中断其他 Pod 或使节点变得不稳定。

由于默认情况下不安全sysctls处于禁用状态，因此 kubelet 不会创建具有不安全sysctl配置文件的 Pod。如果你创建这样的 Pod，调度器会反复将这样的 Pod 分配给节点，而节点却无法启动它。这种无限循环最终会使集群控制平面紧张，使集群变得不稳定。

可以考虑使用 OPA Gatekeeper 或 Kyverno 来拒绝不安全的 Pod。sysctls

## 处理集群升级

自 2021 年 4 月以来，Kubernetes 的发布周期已从每年发布四次（每季度一次）更改为每年发布三次。一个新的次要版本（比如 1. 21 或 1。 22) 大约每十五周发布一次。从 Kubernetes 1.19 开始，每个次要版本在首次发布后将获得大约十二个月的支持。随着 Kubernetes v1.28 的问世，控制平面和工作节点之间的兼容性偏差已从 n-2 扩展到 n-3 次要版本。要了解更多信息，请参阅集群升级的最佳实践。

## 集群终端节点连接

使用 Amazon EKS（弹性 Kubernetes 服务）时，在 Kubernetes 控制平面扩展或修补等事件期间，您可能会遇到连接超时或错误。这些事件可能导致 kube-apiserver 实例被替换，从而可能导致在解析 FQDN 时返回不同的 IP 地址。本文档概述了 Kubernetes API 使用者保持可靠连接的最佳实践。

> **Note**
>
> 实施这些最佳实践可能需要更新客户端配置或脚本，以有效处理新的 DNS 重新解析和重试策略。

主要问题源于 DNS 客户端缓存以及 EKS 端点（公共端点或 X-ENI 私有端点的公共 NLB）的 IP 地址可能过时。替换 kube-apiserver 实例后，完全限定域名 (FQDN) 可能会解析为新的 IP 地址。但是，由于 AWS 托管的 Route 53 区域中的 DNS 生存时间 (TTL) 设置为 60 秒，因此客户可能会在短时间内继续使用过时的 IP 地址。

为了缓解这些问题，Kubernetes API 使用者（例如 kubectl、 CI/CD 管道和自定义应用程序）应实施以下最佳实践：

- 实施 DNS 重新解析
- 使用退避和抖动实现重试。例如，请参阅这篇标题为 “失败发生” 的文章
- 实现客户端超时。设置适当的超时时间，以防止长时间运行的请求阻塞您的应用程序。请注意，某些 Kubernetes 客户端库，尤其是由 OpenAPI 生成器生成的客户端库，可能不允许轻松设置自定义超时。
  - 使用 kubectl 的示例 1：

```
kubectl get pods --request-timeout 10s # default: no timeout
```

  - Python 示例 2：Kubernetes 客户端提供了 _request_ timeout 参数

通过实施这些最佳实践，您可以显著提高应用程序在与 Kubernetes API 交互时的可靠性和弹性。请记住对这些实现进行全面测试，尤其是在模拟故障条件下，以确保它们在实际扩展或修补事件中按预期运行。

### 运行大型集群

EKS 会主动监控控制平面实例的负载，并自动扩展它们以确保高性能。但是，在运行大型集群时，您应该考虑 Kubernetes 中的潜在性能问题和限制，以及 AWS 服务中的配额。

- 根据 ProjectCalico 团队执行的测试，拥有超过 1000 个服务的集群`kube-proxy`在`iptables`模式下使用时可能会遇到网络延迟。解决方案是切换到`kube-proxy`以`ipvs`模式运行。
- 如果 CNI 需要请求 Pod 的 IP 地址，或者您需要经常创建新的 EC2 实例，您也可能会遇到 EC2 API 请求限制。您可以通过将 CNI 配置为缓存 IP 地址来减少 EC2 API 的调用。您可以使用更大的 EC2 实例类型来减少 EC2 扩展事件。

### 其他资源：

- De-mystifying适用于 Amazon EKS 工作节点的集群网络
- Amazon EKS 集群终端节点访问控制
- AWS re: Invent 2019：亚马逊 EKS 幕后 () CON421-R1

## EKS 数据平面

要运行高可用性和弹性的应用程序，您需要一个高可用性和弹性的数据平面。弹性数据平面可确保 Kubernetes 可以自动扩展和修复您的应用程序。弹性数据平面由两个或更多工作节点组成，可以随着工作负载的变化而增长和缩小，并自动从故障中恢复。

使用 EKS 的工作节点有多种选择：EKS 自动模式托管节点、EC2 实例和 Fargat e。

EKS 自动模式提供了通往弹性数据平面的最简单途径。自动模式将 AWS 对 Kubernetes 集群的管理扩展到集群本身之外，让 AWS 还可以设置和管理基础设施，使您的工作负载能够顺利运行。当 Kubernetes 扩展 Pod 时，自动模式会自动向上或向下扩展数据平面，并不断确保集群中的节点大小适当、经济实惠，以适应当前正在运行的工作负载。

如果您选择 EC2 实例，则可以自己管理工作节点或使用 EKS 托管节点组。你可以拥有一个混合了自动模式、托管、自我管理的工作节点和 Fargate 的集群。

Fargate 在隔离的计算环境中运行每个 Pod。在 Fargate 上运行的每个 Pod 都有自己的工作节点。当 Kubernetes 缩放 pod 时，Fargate 会自动缩放数据平面。您可以使用水平容器自动扩缩器来扩展数据平面和工作负载。

扩展 EC2 工作节点的首选方法（如果不使用由 AWS 自动执行的 EKS 自动模式）是使用 Karpenter、Kubernetes 集群自动扩缩器或 EC2 Auto Scaling 组。

## 建议

### 将工作节点和工作负载分布到多个可用区

您可以通过在多个可用区中运行工作节点和 Pod 来保护您的工作负载免受单个可用区出现故障的影响。您可以使用在其中创建节点的子网来控制创建工作节点的可用区。

跨可用区分配 Pod 的推荐方法是对 Pod 使用拓扑分布约束。 Auto-scaling 诸如 EKS Auto Mode 和 Karpenter 之类的功能知道拓扑分布限制，并且会自动在正确的可用区中启动节点以满足您的限制。

如果可能的话，下面的部署将 pod 分布在 AZ 中，如果没有，则让这些 pod 无论如何都能运行：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          whenUnsatisfiable: ScheduleAnyway
          topologyKey: topology.kubernetes.io/zone
          labelSelector:
            matchLabels:
              app: web-server
      containers:
      - name: web-app
        image: nginx
```

```
        resources:
          requests:
            cpu: 1
```

> **Note**
>
> kube-scheduler只能通过带有这些标签的节点来识别拓扑域。如果将上述部署部署部署到仅在单个区域中的节点的集群上，则所有 pod 都将在这些节点上进行调度，因为kube-scheduler他们不知道其他区域。要使此拓扑分布按预期与调度器配合使用，节点必须已存在于所有区域中。拓扑分布约束的minDomains属性用于将符合条件的域的数量通知调度器，即使有节点在那里运行以避免此问题。

> **Warning**
>
> 如果无法whenUnsatisfiable满足拓扑分布约束，则设置为DoNotSchedule将导致 pod 不可调度。只有当 pod 最好不要运行而不是违反拓扑分布约束时，才应设置它。

在旧版本的 Kubernetes 上，你可以使用容器反关联性规则在多个可用区之间调度 Pod。下面的清单告知 Kubernetes 调度器更喜欢在不同的可用区中调度 pod。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      affinity:
        podAntiAffinity:
```

```
        preferredDuringSchedulingIgnoredDuringExecution:
        - podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-server
            topologyKey: failure-domain.beta.kubernetes.io/zone
          weight: 100
    containers:
    - name: web-app
      image: nginx
```

> **Warning**
>
> 不要要求在不同的可用区之间调度 Pod，否则，部署中的 Pod 数量将永远不会超过可用区的数量。

## 使用 EBS 卷时，确保能够在每个可用区中启动节点

如果您使用 Amazon EBS 提供永久卷，则需要确保 Pod 和关联的 EBS 卷位于同一个可用区中。Pod 无法访问位于不同可用区的 EBS-backed 永久卷。Kubernetes 调度器通过节点上的标签知道工作节点位于哪个可用区，并且将始终调度需要与卷位于同一可用区的 EBS 卷的 Pod。但是，如果卷所在的可用区中没有工作节点可用，则无法调度 Pod。

如果使用 EKS 自动模式或 Karpenter，则需要确保在每个可用区 NodeClass 中选择子网。如果使用托管节点组，则需要确保在每个可用区中都有一个节点组。

EBS 存储功能内置于 EKS 自动模式中，但如果使用 Karpenter 或托管节点组，则还需要安装 EBS CSI。

## 使用 EKS 自动模式管理工作节点

EKS Auto Mode 通过为生产就绪型集群提供操作开销最小的操作开销来简化 EKS 的管理。自动模式负责根据集群中运行的 Pod 向上或向下扩展 Node 的数量。节点会自动更新软件补丁和修复，更新将根据配置的中NodePool断设置和 Pod 中断预算执行。

## 运行节点监控代理

节点监控代理通过发布 Kubernetes 事件和更新节点上的状态条件来监控节点运行状况并做出反应。节点监控代理包含在 EKS 自动模式节点中，可以作为 EKS 插件安装，适用于非自动模式管理的节点。

EKS 自动模式、托管节点组和 Karpenter 都能够检测节点监控代理报告的致命节点状况，并在出现这些情况时自动修复这些节点。

## 实施 QoS

对于关键应用程序，可以考虑为 Pod 中的容器定义 `requests = limits`。这将确保在另一个 Pod 请求资源时容器不会被杀死。

最佳做法是对所有容器实施 CPU 和内存限制，因为这样可以防止容器无意中消耗系统资源，从而影响其他共置进程的可用性。

## 为所有工作负载配置和调整资源 Requests/Limits 规模

在调整资源请求和工作负载限制时，可以应用一些一般指导方针：

- 不要指定 CPU 的资源限制。在没有限制的情况下，请求充当容器获得的相对 CPU 时间的权重。这使您的工作负载能够使用完整的 CPU，而不会受到人为限制或饥饿。
- 对于非 CPU 资源，配置 `requests = 可limits`提供最可预测的行为。如果`requests ! =limits`，容器的 QOS 也从 “保证” 降低到 Burstable，因此在节点压力下更有可能被驱逐。
- 对于非 CPU 资源，请勿指定比请求大得多的限制。相对于的配置越大 `limitsrequests`，节点被过度使用的可能性就越大，从而导致工作负载中断的可能性很大。
- 在使用 Karpen ter 或 Cluster 等节点自动缩放解决方案时，正确调整请求大小尤为重要。AutoScaler这些工具会查看您的工作负载请求，以确定要配置的节点的数量和大小。如果你的请求太小，限制更大，那么如果你的工作负载被紧密地挤在节点上，你可能会发现工作负载被驱逐或 OOM 终止。

确定资源请求可能很困难，但是像 Vertic al Pod Autoscaler 这样的工具可以通过在运行时观察容器资源使用情况来帮助你 “调整请求的大小”。可能对确定请求大小有用的其他工具包括：

- Goldilocks
- 帕尔卡
- Prodfiler
- rsg

## 为命名空间配置资源配额

命名空间适用于众多用户分散在多个团队或项目中的环境。它们为名称提供了范围，并且是在多个团队、项目和工作负载之间分配群集资源的一种方式。您可以限制命名空间中的聚合资源消耗。该ResourceQuota对象可以按类型限制可以在命名空间中创建的对象的数量，以及该项目中资源可能消耗的计算资源总量。您可以限制在给定命名空间中可以请求的存储 and/or 计算（CPU 和内存）资源的总和。

如果为计算资源（如 CPU 和内存）启用了命名空间的资源配额，则用户必须为该命名空间中的每个容器指定请求或限制。

考虑为每个命名空间配置配额。考虑使用自动LimitRanges将预配置的限制应用于命名空间内的容器。

## 限制命名空间内的容器资源使用量

资源配额有助于限制命名空间可以使用的资源量。该LimitRange对象可以帮助您实现容器可以请求的最小和最大资源。使用LimitRange可以为容器设置默认请求和限制，如果设置计算资源限制不是组织中的标准做法，这会很有用。顾名思义，LimitRange可以强制命名空间中每个 Pod 或容器的最低和最大计算资源使用量。并且，在命名空间 PersistentVolumeClaim 中强制执行每个存储请求的最小和最大值。

考虑与结合使用 LimitRangeResourceQuota，在容器和命名空间级别强制执行限制。设置这些限制将确保容器或命名空间不会影响集群中其他租户使用的资源。

## 使用 NodeLocal dnsCache

您可以通过运行 NodeLocaldnsCache 来提高集群 DN S 的性能。此功能在群集节点上运行 DNS 缓存代理DaemonSet。所有 pod 都使用节点上运行的 DNS 缓存代理进行名称解析，而不是使用 S kube-dns ervice。此功能自动包含在 EKS 自动模式中。

## 配置自动缩放 CoreDNS

提高集群 DNS 性能的另一种方法是启用 CoreDNS P od 的内置自动缩放功能。

此功能持续监控集群状态，包括节点数量和 CPU 内核。控制器会根据这些信息，动态调整 EKS 集群中 CoreDNS 部署的副本数量。

# 联网最佳实践

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

了解 Kubernetes 网络对于高效运行集群和应用程序至关重要。Pod 网络，也称为集群网络，是 Kubernetes 网络的中心。Kubernetes 支持用于集群联网的容器网络接口 (CNI) 插件。

亚马逊 EKS 正式支持亚马逊虚拟私有云 (VPC) Virtual Privat e CNI 插件来实现 Kubernetes Pod 联网。VPC CNI 提供与 AWS VPC 的原生集成，并在底层模式下工作。在底层模式下，Pod 和主机位于同一网络层并共享网络命名空间。从集群和 VPC 的角度来看，Pod 的 IP 地址是一致的。

本指南介绍了 Kubernet es 集群联网背景下的亚马逊 VPC 容器网络接口 (VPC CNI)。VPC CNI 是 EKS 支持的默认网络插件，因此是本指南的重点。VPC CNI 具有高度可配置性，可支持不同的用例。本指南还包括关于不同 VPC CNI 用例、操作模式、子组件的专门章节，然后是建议。

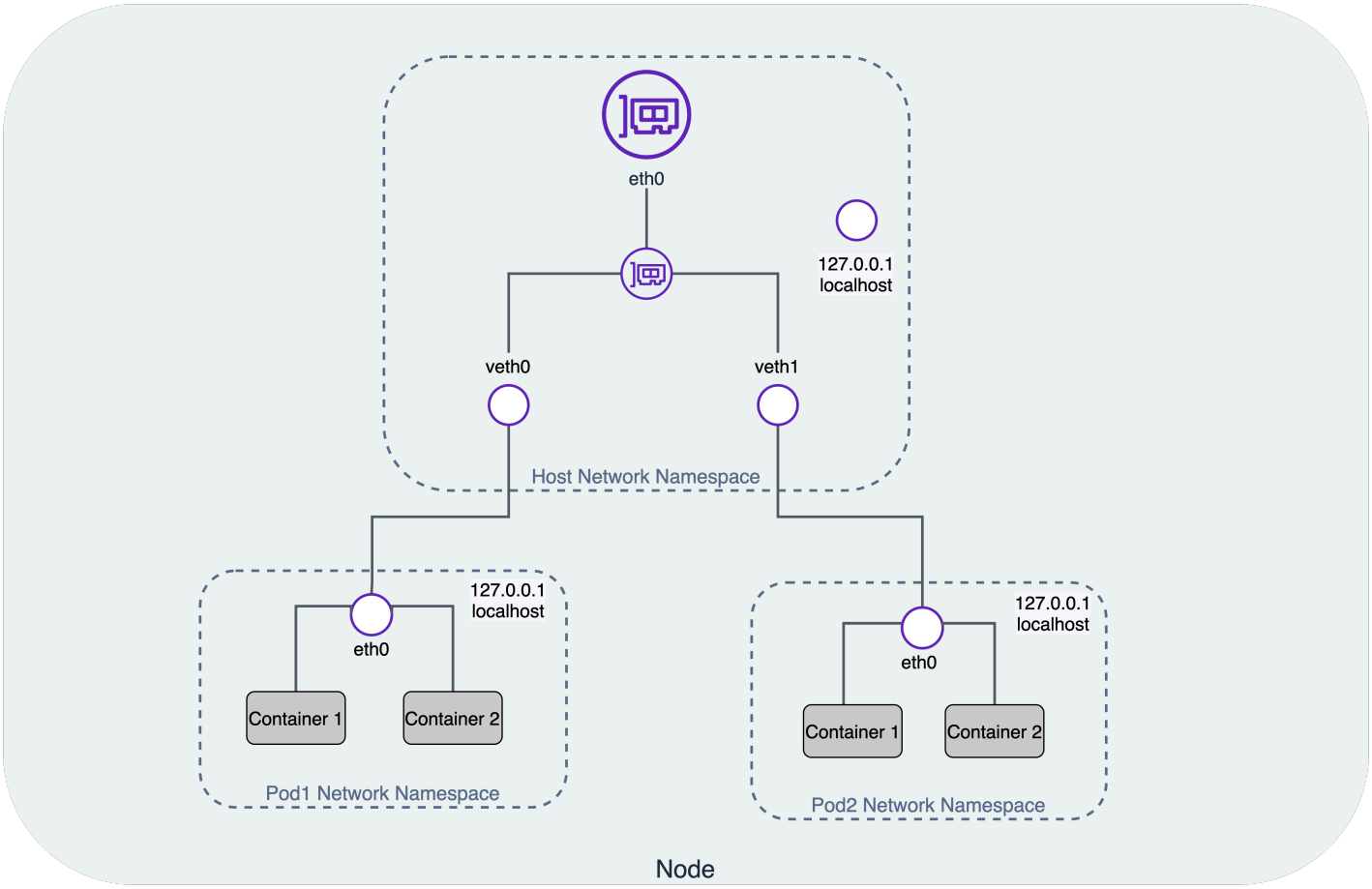
亚马逊 EKS 在上游 Kubernetes 上游运行，并已通过 Kubernetes 合规认证。尽管您可以使用备用 CNI 插件，但本指南并未提供管理备用 CNI 的建议。查看 EKS 备用 CNI 文档，了解有效管理备用 CNI 的合作伙伴和资源列表。

## Kubernetes 网络模型

Kubernetes 对集群网络设置了以下要求：

- 在同一节点上调度的 Pod 必须能够在不使用 NAT（网络地址转换）的情况下与其他 Pod 通信。
- 在特定节点上运行的所有系统守护程序（后台进程，例如 kubelet）都可以与在同一节点上运行的 Pod 通信。
- 使用主机网络的 Pod 必须能够在不使用 NAT 的情况下联系所有其他节点上的所有其他 Pod。

有关 Kubernetes 对兼容网络实现的期望的详细信息，请参阅 Kubernetes 网络模型。下图说明了 Pod 网络命名空间和主机网络命名空间之间的关系。




# 容器网络接口 (CNI)

Kubernetes 支持 CNI 规范和插件来实现 Kubernetes 网络模型。CNI 由规范（当前版本 1.0.0）和用于编写用于在容器中配置网络接口的插件的库以及许多支持的插件组成。CNI 只关心容器的网络连接以及在删除容器时移除分配的资源。

通过向 kubelet 传递命令行选项即可启用 CNI 插件。`--network-plugin=cni`Kubelet 从`--cni-conf-dir`（默认为 etc/cni //net.d）读取一个文件，并使用该文件中的 CNI 配置来设置每个 Pod 的网络。CNI 配置文件必须符合 CNI 规范（最低版本 0.4.0），并且该配置引用的任何必需的 CNI 插件都必须存在于`--cni-bin-dir`目录中（默认为 //bin）。opt/cni如果目录中有多个 CNI 配置文件，kubelet 会使用按字典顺序排在首位的配置文件。

# 亚马逊虚拟私有云 (VPC) Virtual Private Cloud CNI

AWS-provided VPC CNI 是 EKS 集群的默认联网附加组件。在您配置 EKS 集群时，会默认安装 VPC CNI 插件。VPC CNI 在 Kubernetes 工作节点上运行。VPC CNI 插件由 CNI 二进制文件和 IP 地址管

理 (ipamd) 插件组成。CNI 将来自 VPC 网络的 IP 地址分配给 Pod。ipamd 管理每个 Kubernetes 节点的 AWS 弹性网络接口 (ENI)，并维护热池 IP。VPC CNI 提供配置选项，用于预先分配 ENI 和 IP 地址，从而缩短 Pod 启动时间。有关推荐的插件管理最佳实践，请参阅 Amazon VPC CNI。

Amazon EKS 建议您在创建集群时至少在两个可用区中指定子网。Amazon VPC CNI 从节点子网为 Pod 分配 IP 地址。我们强烈建议您检查子网中是否有可用的 IP 地址。在部署 EKS 集群之前，请考虑 VPC 和子网建议。

Amazon VPC CNI 从连接到节点主 ENI 的子网中分配一个由 ENI 和辅助 IP 地址组成的温池。这种 VPC CNI 模式称为辅助 IP 模式。IP 地址的数量以及相应的 Pod 数量（Pod 密度）由 ENI 的数量和实例类型定义的每个 ENI 的 IP 地址（限制）来定义。辅助模式是默认模式，适用于实例类型较小的小型集群。如果您遇到吊舱密度问题，请考虑使用前缀模式。您还可以通过为 ENI 分配前缀来增加 Pod 节点上的可用 IP 地址。

Amazon VPC CNI 以原生方式与 AWS VPC 集成，允许用户应用现有的 AWS VPC 联网和安全最佳实践来构建 Kubernetes 集群。这包括能够使用 VPC 流日志、VPC 路由策略和安全组进行网络流量隔离。默认情况下，Amazon VPC CNI 会将与节点上的主 ENI 关联的安全组应用于 Pod。如果您想为 Pod 分配不同的网络规则，可以考虑为 Pod 启用安全组。

默认情况下，VPC CNI 会将分配给节点主 ENI 的子网中的 Pod 分配 IP 地址。运行包含数千个工作负载的大型集群时，通常会遇到 IPv4 地址短缺的情况。AWS VPC 允许您通过分配辅助 CIDR 来解决 IPv4 CIDR 块耗尽的问题，从而扩展可用的 IP。AWS VPC CNI 允许您为 Pod 使用不同的子网 CIDR 范围。VPC CNI 的这一功能称为自定义联网。您可以考虑将自定义网络与 EKS `100.64.0.0/10` 范围中的 CIDR（共享地址空间，RFC 6598）一起使用。这实际上允许您创建一个环境，让 Pod 不再使用您的 VPC 中的任何 RFC1918 IP 地址。

自定义网络是解决 IPv4 地址耗尽问题的一种选择，但它需要运营开销。我们建议使用 IPv6 群集而不是自定义网络来解决此问题。具体而言，如果您已完全用完 VPC 的所有可用 IPv4 地址空间，我们建议您迁移到 IPv6 集群。评估贵组织支持 IPv6 的计划，并考虑投资 IPv6 是否具有更高的长期价值。

EKS 对 IPv6 的支持侧重于解决由 IPv4 地址空间有限引起的 IP 耗尽问题。为了应对客户的 IPv4 耗尽问题，EKS 将 Pod 置于双堆栈 IPv6-only Pod 之上。也就是说，Pod 可能能够访问 IPv4 资源，但没有从 VPC CIDR 范围为它们分配 IPv4 地址。VPC CNI 从 AWS 托管的 VPC IPv6 CIDR 块中为 Pod 分配 IPv6 地址。

## 子网计算器

该项目包括子网计算器 Excel 文档。本计算器文档模拟了不同的 ENI 配置选项（例如`WARM_IP_TARGET`和`WARM_ENI_TARGET`）下指定工作负载的 IP 地址消耗情况。该文档包括两张

纸，第一页用于热 ENI 模式，另一页用于暖 IP 模式。有关这些模式的更多信息，请查看 VPC CNI 指南。

输入：

- 子网 CIDR 大小
- 热 ENI 目标或热 IP 目标
- 实例列表
  - 每个实例调度的工作负载 Pod 的类型、数量和数量

输出：

- 托管的 pod 总数
- 消耗的子网 IP 数量
- 剩余子网 IP 的数量
- 实例级别详情
  - IPs/ENIs 每个实例的 Warm 数量
  - IPs/ENIs 每个实例的活动数量

# VPC 和子网注意事项

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

运营 EKS 集群需要了解 AWS VPC 联网以及 Kubernetes 联网知识。

我们建议您在开始设计 VPC 或在现有 VPC 中部署集群之前，先了解 EKS 控制平面通信机制。

在构建 VPC 和子网以与 EKS 配合使用时，请参阅集群 VPC 注意事项和 Amazon EKS 安全组注意事项。

# 概述

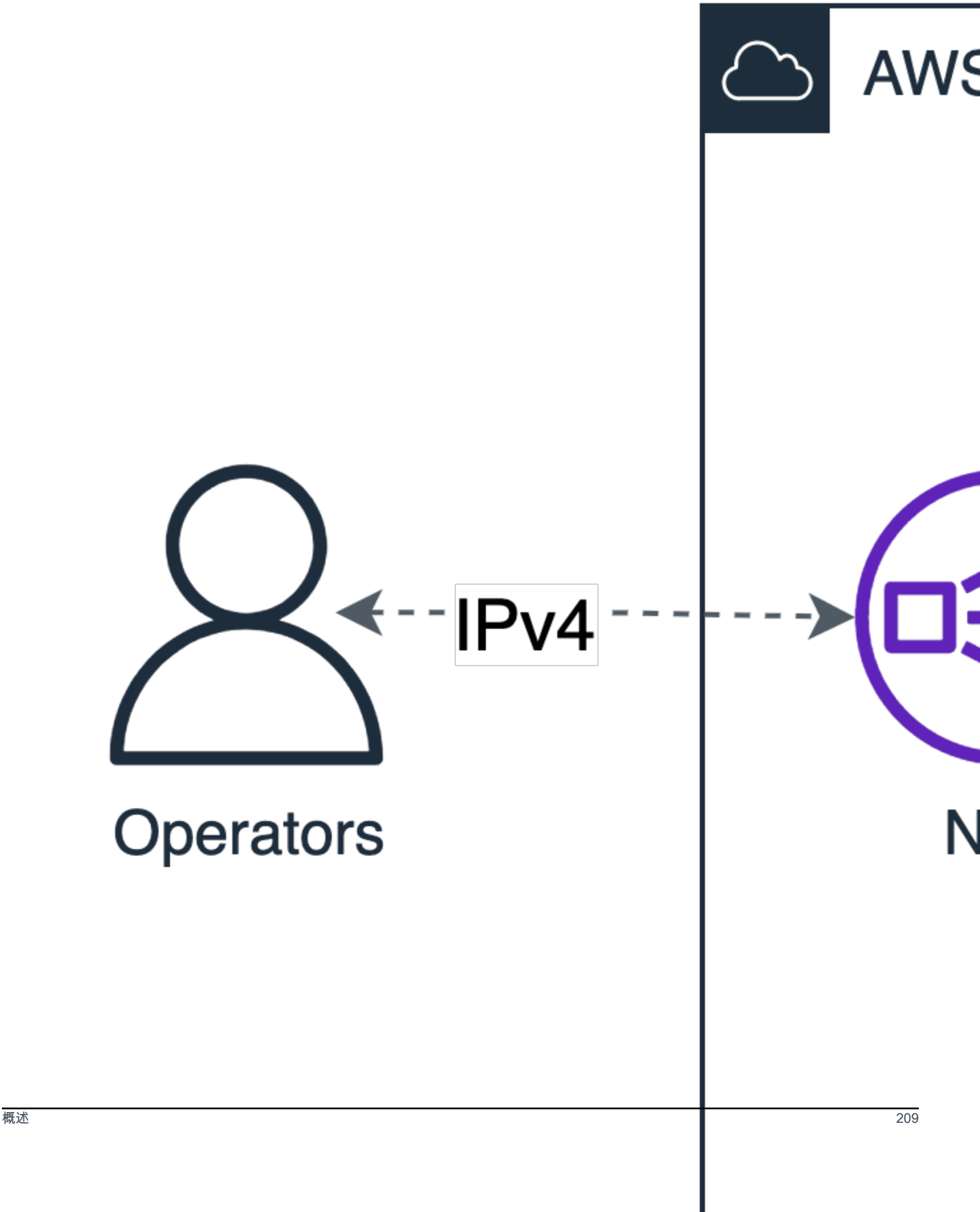
## EKS 集群架构

一个 EKS 集群由两个 VPC 组成：

- 托管 Kubernetes 控制平面的 AWS-managed VPC。此 VPC 不会出现在客户账户中。
- 托管 Kubernetes 节点的由客户管理的 VPC。这是容器以及其他客户管理的 AWS 基础设施（例如集群使用的负载均衡器）的运行场所。此 VPC 出现在客户账户中。在创建集群之前，您需要创建客户托管的 VPC。如果您未提供 VPC，则 eksctl 会创建一个 VPC。

客户 VPC 中的节点需要能够连接到 AWS VPC 中的托管 API 服务器终端节点。这允许节点在 Kubernetes 控制平面上注册并接收运行应用程序 Pod 的请求。

这些节点通过 (a) EKS 公共端点或 (b) 由 EKS 管理的 Cross-Account 弹性网络接口 (X-ENI) 连接到 EKS 控制平面。创建集群时，您需要指定至少两个 VPC 子网。EKS X-ENI 在创建集群期间指定的每个子网（也称为集群子网）中放置一个。Kubernetes API 服务器使用这些 Cross-Account ENI 与部署在客户管理的集群 VPC 子网上的节点进行通信。


AWS
IPv4
Operators
N

节点启动时，将执行 EKS 引导脚本并安装 Kubernetes 节点配置文件。作为每个实例启动过程的一部分，将启动容器运行时代理、kubelet 和 Kubernetes 节点代理。

要注册节点，Kubelet 会联系 Kubernetes 集群终端节点。它与 VPC 外部的公共终端节点或 VPC 内的私有终端节点建立连接。Kubelet 接收 API 指令，并定期向端点提供状态更新和心跳。

## EKS 控制平面通信

EKS 有两种方法可以控制对集群终端节点的访问。终端节点访问控制允许您选择是可以从公共 Internet 访问终端节点，还是只能通过您的 VPC 访问终端节点。您可以同时打开公共终端节点（这是默认值）、私有终端节点，或者同时开启两者。

集群 API 终端节点的配置决定了节点与控制平面通信所采用的路径。请注意，这些终端节点设置可以随时通过 EKS 控制台或 API 进行更改。

公共终端节点

这是新 Amazon EKS 集群的默认行为。当只启用集群的公共终端节点时，来自集群 VPC（例如控制平面通信的工作节点）内的 Kubernetes API 请求会离开 VPC，但不会离开 Amazon 的网络。为了使节点能够连接到控制平面，它们必须具有公有 IP 地址和通往 Internet 网关的路由，或者可以使用 NAT 网关的公有 IP 地址的 NAT 网关的路由。

公有和私有端点

启用公有和私有终端节点后，来自 VPC 内部的 Kubernetes API 请求将通过 X-ENIs 您的 VPC 内部与控制平面进行通信。集群 API 服务器可从 Internet 访问。

私有端点

如果仅启用私有端点，则无法通过互联网公开访问您的 API 服务器。传输到集群 API 服务器的所有流量都必须来自您的集群的 VPC 或连接的网络中。这些节点通过您的 VPC X-ENIs 内部与 API 服务器通信。请注意，集群管理工具必须有权访问私有终端节点。详细了解如何从 Amazon VPC 外部连接到私有 Amazon EKS 集群终端节点。

请注意，集群的 API 服务器终端节点由公共 DNS 服务器解析为来自 VPC 的私有 IP 地址。过去，终端节点只能在 VPC 内部解析。

## VPC 配置

Amazon VPC 同时支持 IPv4 和 IPv6 寻址。亚马逊 EKS 默认支持 IPv4。VPC 必须具有一个与之关联的 IPv4 CIDR 块。您可以选择将多个 IPv4 无类 Inter-Domain 路由 (CIDR) 块和多个 IPv6 CIDR 块关

联到您的 VPC。创建 VPC 时，必须按照 RFC 1918 中指定的私有 IPv4 地址范围为该 VPC 指定 IPv4 网段。允许的区块大小介于/16前缀（65,536 个 IP 地址）和/28前缀（16 个 IP 地址）之间。

创建新 VPC 时，您可以连接一个 IPv6 CIDR 块，在更改现有 VPC 时最多可以连接五个。IPv6 CIDR 块大小的前缀长度可以介于 /44 和 /60 之间，对于 IPv6 子网，前缀长度可以介于 /44/ 和 /64 之间。您可以从亚马逊维护的 IPv6 地址池中请求 IPv6 CIDR 块。有关更多信息，请参阅《VPC 用户指南》的 VPC CIDR 块部分。

亚马逊 EKS 集群同时支持 IPv4 和 IPv6。默认情况下，EKS 集群使用 IPv4 IP。在创建集群时指定 IPv6 将启用使用 IPv6 集群。IPv6 集群需要双堆栈 VPC 和子网。

Amazon EKS 建议您在创建集群时至少使用两个位于不同可用区域的子网。您在创建集群时传入的子网称为集群子网。创建集群时，Amazon EKS 会在您指定的子网中创建最多 4 个跨账户（x 账户或 X-enis）网卡。X-enis 始终处于部署状态，用于集群管理流量，例如日志传输、exec 和代理。请参阅 EKS 用户指南，了解完整的 VPC 和子网要求详情。

Kubernetes 工作节点可以在集群子网中运行，但不建议这样做。在集群升级期间，Amazon EKS 会在集群子网中配置其他 ENI。当您的集群向外扩展时，工作节点和 Pod 可能会消耗集群子网中的可用 IP。因此，为了确保有足够的可用IP，您可能需要考虑使用带有 /28 网络掩码的专用集群子网。

Kubernetes 工作节点可以在公有子网或私有子网中运行。子网是公有子网还是私有子网是指子网内的流量是通过互联网网关路由的。公有子网有通过互联网网关进入互联网的路由表入口，但私有子网没有。

源自其他地方并到达您的节点的流量称为入口。来自节点并离开网络的流量称为出口。在配置了互联网网关的子网中具有公有或弹性 IP 地址 (EIP) 的节点允许从 VPC 外部进入。私有子网通常具有到 NAT 网关的路由，这不允许从 VPC 外部进入子网中节点的入口流量，同时仍允许来自节点的流量离开 VPC（出口）。

在 IPv6 世界中，每个地址都可以通过互联网进行路由。与节点和 Pod 关联的 IPv6 地址是公开的。通过在 VPC 中实现仅限出口的互联网网关 (EIGW) 来支持私有子网，允许出站流量，同时阻止所有传入流量。在 V PC 用户指南中可以找到实施 IPv6 子网的最佳实践。

### 您可以通过三种不同的方式配置 VPC 和子网：

仅使用公有子网

在同一个公共子网中，同时创建节点和入口资源（例如负载均衡器）。使用标记公有子网`kubernetes.io/role/elb`以构造面向互联网的负载均衡器。在此配置中，可以将集群终端节点配置为公共终端节点、私有终端节点或两者兼而有之（公有和私有）。

使用私有子网和公有子网

节点是在私有子网上创建的，而 Ingress 资源是在公共子网中实例化的。您可以启用对集群终端节点的公共、私有或两者（公有和私有）访问权限。根据集群终端节点的配置，节点流量将通过 NAT 网关或 ENI 进入。

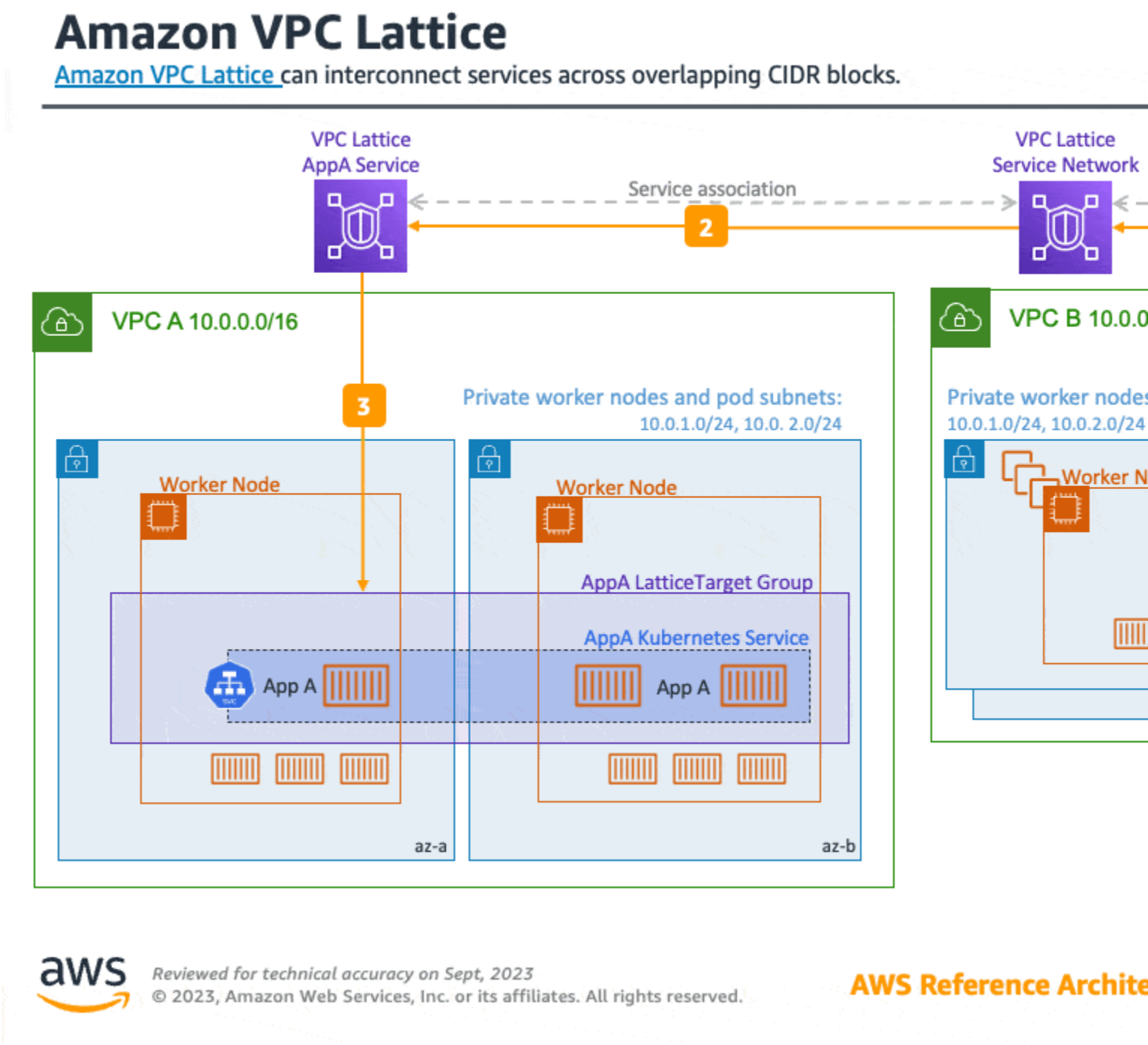
仅使用私有子网

节点和入口都是在私有子网中创建的。使用子kubernetes.io/role/internal-elb网标签来构造内部负载均衡器。访问集群的终端节点将需要 VPN 连接。您必须激活 AWS f PrivateLink or EC2 以及所有 Amazon ECR 和 S3 存储库。只应启用集群的私有终端节点。我们建议在配置私有集群之前仔细阅读 EKS 私有集群要求。

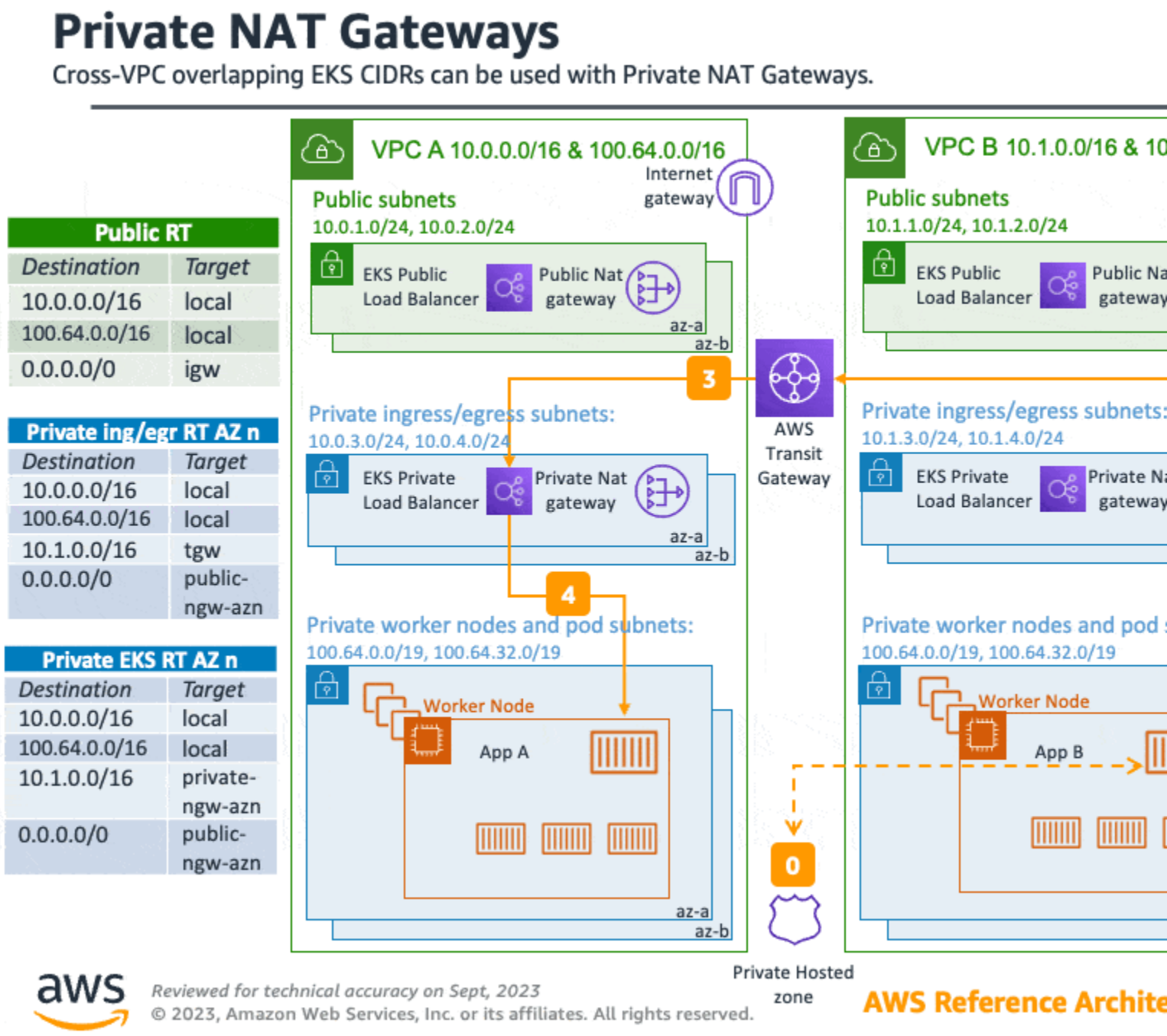
## 在 VPC 之间进行通信

在许多情况下，您需要将多个 VPC 和单独的 EKS 集群部署到这些 VPC。

您可以使用 Amazon VPC Lattice 在多个 VPC 和账户之间持续安全地连接服务（无需通过 VPC 对等互连、AWS PrivateLink 或 AWS Transit Gateway 等服务提供额外的连接）。点击此处了解更多信息。




Amazon VPC Lattice 在 IPv4 和 IPv6 的本地链路地址空间中运行，为可能具有重叠的 IPv4 地址的服务提供连接。为了提高运营效率，我们强烈建议将 EKS 集群和节点部署到不重叠的 IP 范围。如果您的基础设施包含 IP 范围重叠的 VPC，则需要相应地架构您的网络。我们建议私有 NAT 网关或自定义联网模式下的 VPC CNI 与传输网关结合使用，在 EKS 上集成工作负载，以解决重叠的 CIDR 挑战，同时保留可路由的 RFC1918 IP 地址。

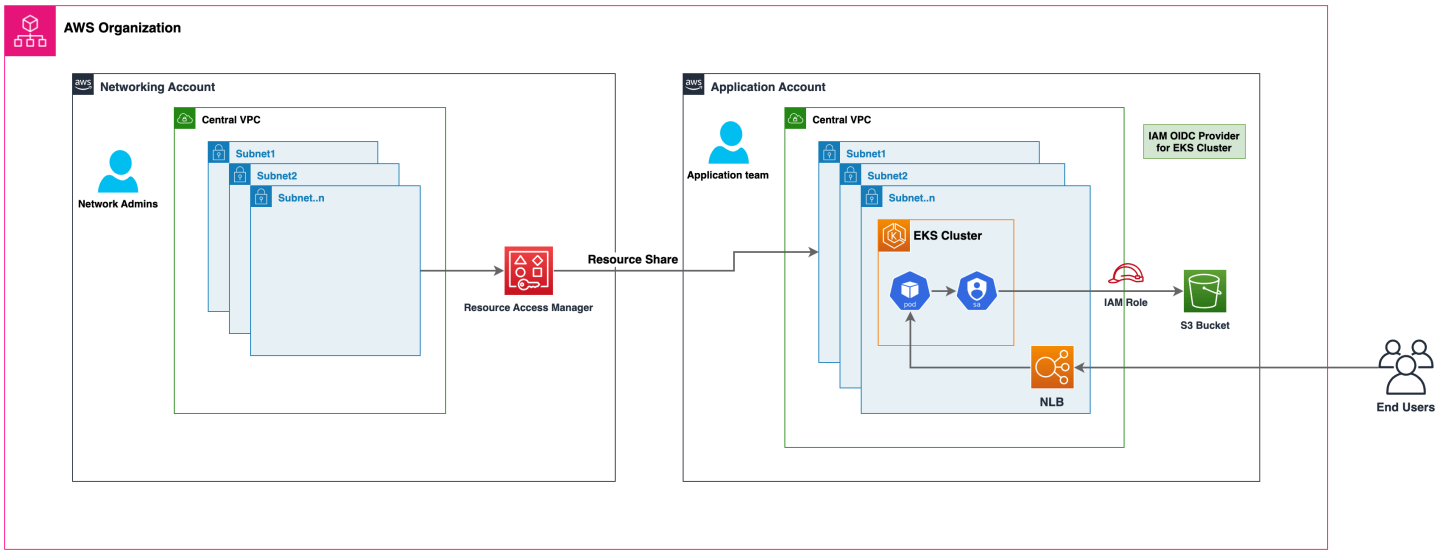



如果您是服务提供商 PrivateLink，并且希望在不同的账户中与您的客户 VPC 共享您的 Kubernetes 服务和入口（ALB 或 NLB），请考虑使用 AWS（也称为终端节点服务）。

# 跨多个账户共享 VPC

许多企业采用共享的 Amazon VPC 来简化网络管理、降低成本并提高 AWS 组织中多个 AWS 账户的安全性。他们利用 AWS Resource Access Manager (RAM) 与个人 AWS 账户、组织单位 (OU) 或整个 AWS 组织安全地共享支持的 AWS 资源。

您可以使用 AWS RAM 从另一个 AWS 账户在共享 VPC 子网中部署 Amazon EKS 集群、托管节点组和其他支持 AWS 资源（如 LoadBalancers安全组、端点等）。下图描绘了一个高级架构示例。这允许中央网络团队控制 VPC、子网等网络结构，同时允许应用程序或平台团队在各自的 AWS 账户中部署 Amazon EKS 集群。此场景的完整演练可在此 github 存储库中找到。




使用共享子网时的注意事项

- Amazon EKS 集群和工作节点可以在属于同一 VPC 的共享子网中创建。Amazon EKS 不支持跨多个 VPC 创建集群。
- Amazon EKS 使用 AWS VPC 安全组 (SG) 来控制 Kubernetes 控制平面和集群工作节点之间的流量。安全组还用于控制工作节点与其他 VPC 资源以及外部 IP 地址之间的流量。您必须在 application/participant 账户中创建这些安全组。确保您打算用于 pod 的安全组也位于参与者账户中。您可以在安全组中配置入站和出站规则，以允许进出位于 Central VPC 账户中的安全组的必要流量。
- 在您的 Amazon EKS 集群所在的参与者账户中创建 IAM 角色和相关策略。这些 IAM 角色和策略对于向由 Amazon EKS 管理的 Kubernetes 集群以及在 Fargate 上运行的节点和容器授予必要权限至关重要。这些权限允许 Amazon EKS 代表您调用其他 AWS 服务。
- 您可以按照以下方法允许跨账户从 k8s 容器访问 AWS 资源，例如 Amazon S3 存储桶、Dynamodb 表等：

  - 基于资源的策略方法：如果 AWS 服务支持资源策略，则可以添加适当的基于资源的策略，以允许跨账户访问分配给 kubernetes pod 的 IAM 角色。在这种情况下，OIDC 提供商、IAM 角色和权限策略存在于应用程序账户中。要查找支持基于资源的策略的 AWS 服务，请参阅与 IAM 配合使用的 A WS 服务，然后在 “基于资源” 列中查找标有 “是” 的服务。
  - OIDC 提供商方法：OIDC 提供商、IAM 角色、权限和信任策略等 IAM 资源将在资源所在的其他参与者 AWS 账户中创建。这些角色将分配给应用程序账户中的 Kubernetes pod，以便它们可以访问跨账户资源。有关此方法的完整演练，请参阅 Kubernetes 服务账户的跨账户 IAM 角色博客。
- 您可以部署 Amazon 弹性负载均衡器 (ELB) 资源（ALB 或 NLB），将流量路由到应用程序账户或中央网络账户中的 k8s 容器。有关在中央网络账户中部署 ELB 资源的详细说明，请参阅通过 Cross-Account 负载均衡器公开 Amazon EKS Pod 演练。此选项提供了更高的灵活性，因为它允许中央网络账户完全控制 Load Balancer 资源的安全配置。
- 使用 `custom networking feature` Amazon VPC CNI 时，您需要使用中央网络账户中列出的可用区 (AZ) ID 映射来创建每个映射。`ENIConfig`这是因为物理可用区随机映射到每个 AWS 账户中的可用区名称。

### 安全组

安全组控制允许到达和离开与其关联资源的流量。Amazon EKS 使用安全组来管理控制平面和节点之间的通信。在创建集群时，Amazon EKS 将创建一个名为 `eks-cluster-sg-my-cluster-uniqueID` 的安全组。EKS 将这些安全组关联到托管 ENI 和节点。默认规则允许所有流量在集群和节点之间自由流动，并允许所有出站流量到任何目的地。

创建集群时，您可以指定自己的安全组。当您指定自己的安全组时，请参阅有关安全组的建议。

## 建议

### 考虑 Multi-AZ 部署

AWS 区域提供多个物理分隔和隔离的可用区 (AZ)，这些可用区通过低延迟、高吞吐量和高度冗余的网络相连。借助可用区，您可以设计和操作在可用区之间自动进行故障切换而不会中断的应用程序。Amazon EKS 强烈建议将 EKS 集群部署到多个可用区。创建集群时，请考虑在至少两个可用区中指定子网。

在节点上运行的 Kubelet 会自动为节点对象添加标签，例如。`topology.kubernetes.io/region=us-west-2`我们建议将节点标签与 Pod 拓扑分布约束结合使用，以控制 Pod 的跨区域分布方式。这些提示使 Kubernetes 调度器能够放置 Pod 以获得更好的预期可用性，从而降低相关故障影响整个工作负载的风险。请参阅为 Pod 分配节点，查看节点选择器和可用区分布限制的示例。

您可以在创建节点时定义子网或可用区。如果未配置子网，则节点将放置在集群子网中。EKS 对托管节点组的支持会根据可用容量自动将节点分布在多个可用区域中。如果工作负载定义了拓扑分布限制，Karpenter 将通过将节点扩展到指定可用区来实现可用区分布布局。

AWS 弹性负载均衡器由 Kubernetes 集群的 AWS Load Balancer 控制器管理。它为 Kubernetes 入口资源配置应用程序负载均衡器 (ALB)，为负载均衡器类型的 Kubernetes 服务预置网络负载均衡器 (NLB)。Elastic Load Balancer 控制器使用标签来发现子网。ELB 控制器至少需要两个可用区 (AZ) 才能成功配置入口资源。考虑在至少两个可用区中设置子网，以利用地理冗余的安全性和可靠性。

### 将节点部署到私有子网

包含私有子网和公有子网的 VPC 是在 EKS 上部署 Kubernetes 工作负载的理想方法。考虑在两个不同的可用区中至少设置两个公有子网和两个私有子网。公有子网的相关路由表包含通往 Internet 网关的路由。Pod 能够通过 NAT 网关与互联网进行交互。在 IPv6 环境 (EI GW) 中，仅限出口的互联网网关支持私有子网。

在私有子网中实例化节点可以最大限度地控制节点流量，并且对绝大多数 Kubernetes 应用程序有效。入口资源（如负载均衡器）在公共子网中实例化，并将流量路由到在私有子网上运行的 Pod。

如果您要求严格的安全性和网络隔离，请考虑使用仅限私有模式。在此配置中，三个私有子网部署在 AWS 区域的 VPC 内的不同可用区中。部署到子网的资源无法访问互联网，互联网也无法访问子网中的资源。为了使您的 Kubernetes 应用程序能够访问其他 AWS 服务，您必须配置 PrivateLink 接口网 and/or 关终端节点。您可以设置内部负载均衡器，使用 AWS Load Balancer Controller 将流量重定向到 Pod。必须对私有子网进行标记 (`kubernetes.io/role/internal-elb: 1`)，控制器才能配置负载均衡器。要在集群中注册节点，必须将集群终端节点设置为私有模式。请访问私有集群指南，了解完整的要求和注意事项。

### 考虑集群终端节点的公用和私有模式

Amazon EKS 提供仅限公开、公有和私有以及仅限私有的集群终端节点模式。默认模式为仅限公开，但我们建议将集群终端节点配置为公共模式和私有模式。此选项允许集群 VPC 内的 Kubernetes API 调用（例如节点到控制平面通信）使用私有 VPC 终端节点，流量将流量保留在集群的 VPC 中。另一方面，可以从互联网访问您的集群 API 服务器。但是，我们强烈建议限制可以使用公共终端节点的 CIDR 块。了解如何配置公有和私有终端节点访问权限，包括限制 CIDR 块。

当您需要安全性和网络隔离时，我们建议您使用仅限私有的端点。我们建议使用 E KS 用户指南中列出的任一选项私下连接到 API 服务器。

### 谨慎配置安全组

Amazon EKS 支持使用自定义安全组。任何自定义安全组都必须允许节点与 Kubernetes 控制平面之间的通信。当您的组织不允许开放通信时，请检查端口要求并手动配置规则。

EKS 会将您在集群创建期间提供的自定义安全组应用于托管接口（X-ENIs）。但是，它不会立即将它们与节点相关联。在创建节点组时，强烈建议手动关联自定义安全组。请考虑启用安全性 GroupSelectorTerms，以便在节点自动缩放期间启用自定义安全组的 Karpenter 节点模板发现。

我们强烈建议创建一个安全组以允许所有节点间通信流量。在引导过程中，节点需要出站 Internet 连接才能访问集群终端节点。评估外部访问要求，例如本地连接和容器注册表访问权限，并适当设置规则。在将更改投入生产之前，我们强烈建议您在开发环境中仔细检查连接。

### 在每个可用区部署 NAT 网关

如果您在私有子网（IPv4 和 IPv6）中部署节点，请考虑在每个可用区 (AZ) 中创建一个 NAT 网关，以确保架构独立于区域并减少跨可用区的开支。可用区中的每个 NAT 网关都采用冗余实现。

## Amazon VPC CNI

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Amazon EKS 通过亚马逊 VPC 容器网络接口插件（也称为 VPC CNI）实现集群联网。CNI 插件允许 Kubernetes Pod 拥有与 VPC 网络上相同的 IP 地址。更具体地说，Pod 内的所有容器共享一个网络命名空间，它们可以使用本地端口相互通信。

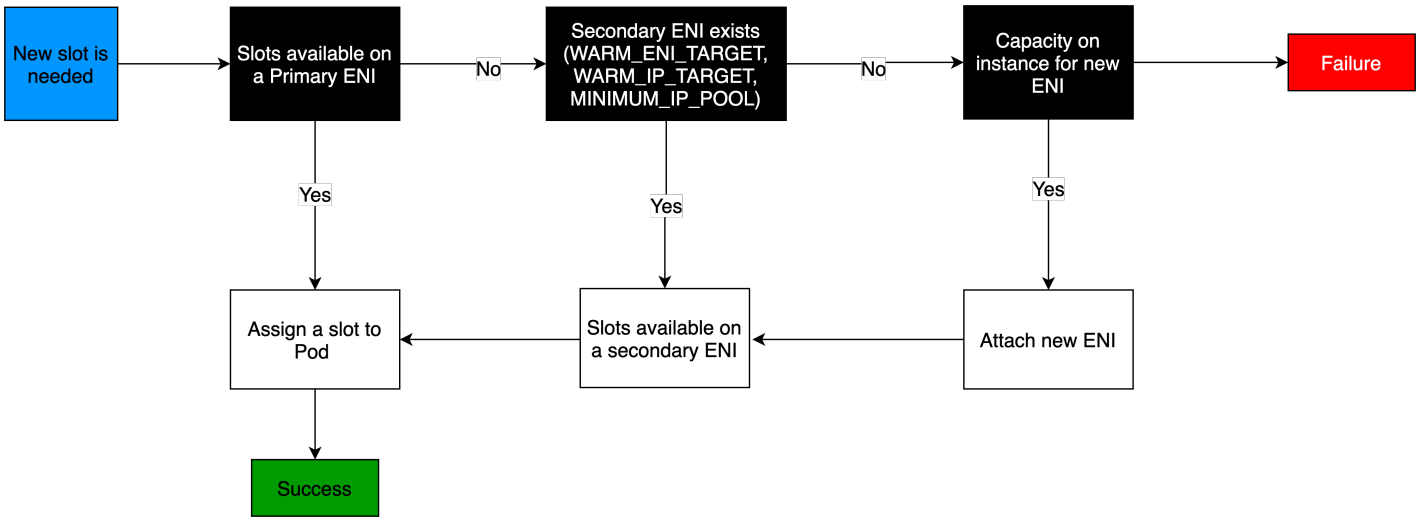
Amazon VPC CNI 有两个组成部分：

- CNI Binary，它将设置 Pod 网络以启用 Pod-to-Pod 通信。CNI 二进制文件在节点根文件系统上运行，当向节点添加新 Pod 或从节点中移除现有 Pod 时，kubelet 会调用该二进制文件。
- ipamd，一个长期运行的节点本地 IP 地址管理 (IPAM) 守护程序，负责：
  - 管理节点上的 ENI，以及
  - 维护可用 IP 地址或前缀的热池

创建实例时，EC2 会创建并连接与主子网关联的主网卡。主子网可以是公有子网或私有子网。在 HostNetwork 模式下运行的 Pod 使用分配给节点主 ENI 的主要 IP 地址，并且与主机共享相同的网络命名空间。

CNI 插件管理节点上的弹性网络接口 (ENI)。配置节点后，CNI 插件会自动将节点子网中的插槽池（IP 或 Prefix）分配给主 ENI。此池称为温池，其大小由节点的实例类型决定。根据 CNI 设置，插槽可能是 IP 地址或前缀。在 ENI 上分配插槽后，CNI 可能会将带有温插槽池的其他 ENI 连接到节点。这些额外的弹性网卡被称为次要弹性网卡。根据实例类型，每个 ENI 只能支持一定数量的插槽。CNI 根据

所需的插槽数量（通常与 Pod 的数量相对应）将更多 ENI 附加到实例。此过程一直持续到节点无法再支持其他 ENI 为止。CNI 还预先分配 “热” ENI 和插槽，以便更快地启动 Pod。请注意，每种实例类型都有可连接的最大弹性网卡数量。除了计算资源外，这是 Pod 密度（每个节点的 Pod 数量）的一个限制。




您可以使用的网络接口的最大数量和最大插槽数因 EC2 实例的类型而异。由于每个 Pod 在某个插槽上消耗一个 IP 地址，因此您可以在特定 EC2 实例上运行的 Pod 数量取决于可以连接多少 ENI 以及每个 ENI 支持多少插槽。我们建议按照 EKS 用户指南设置最大 Pod 数，以避免耗尽实例的 CPU 和内存资源。使用`hostNetwork`的 Pod 不在此计算范围内。有关更多信息，请参阅 Amazon EKS 用户指南中的如何确定 MaxPods。

## 概述

辅助 IP 模式是 VPC CNI 的默认模式。本指南概述了启用辅助 IP 模式时的 VPC CNI 行为。ipamd（IP 地址分配）的功能可能会有所不同，具体取决于 VPC CNI 的配置设置，例如the section called “适用于 Linux 的前缀模式”the section called “每个 Pod 的安全组”、和。the section called “自定义网络”

Amazon VPC CNI 作为名为 aws-node 的 Kubernetes 守护程序部署在工作节点上。配置工作节点时，它会附加一个默认 ENI，称为主 ENI。CNI 从连接到节点主 ENI 的子网中分配一个 ENI 和辅助 IP 地址的温池。默认情况下，ipamd 会尝试向该节点分配额外的 ENI。当调度单个 Pod 并从主 ENI 分配辅助 IP 地址时，IPAMD 会分配额外的 ENI。这种 “温暖” ENI 可实现更快的 Pod 联网。随着辅助 IP 地址池的用完，CNI 会添加另一个 ENI 来分配更多。

池中的 ENI 和 IP 地址数量是通过名为 WARM_ENI_TARGET、WARM_IP_TARGET、MINIMUM_IP_TARGET 的环境变量配置的。`aws-node`守护程序将定期检查是否连接了足够数量的 ENI。当满足所有`WARM_ENI_TARGET`、

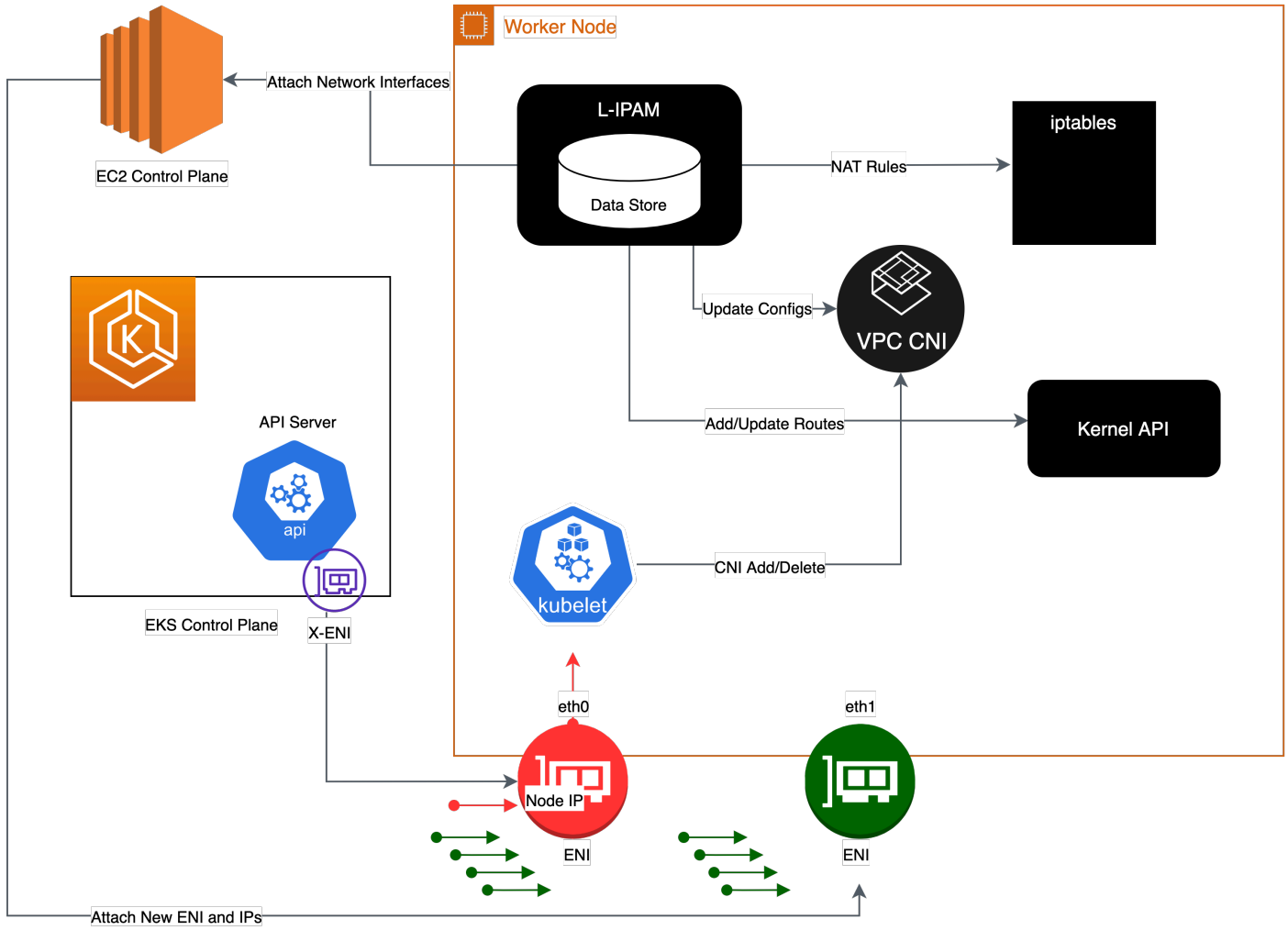
或WARM_IP_TARGET和MINIMUM_IP_TARGET条件时，就会附上足够数量的 ENI。如果连接的 ENI 不足，CNI 将向 EC2 发出 API 调用以附加更多 ENI，直到MAX_ENI达到限制。

- WARM_ENI_TARGET-整数，大于 0 的值表示已启用要求
  - 要维护的热弹性网卡数量。当 ENI 作为辅助 ENI 连接到节点时，它是 “温暖的”，但任何 Pod 均未使用它。更具体地说，未将 ENI 的 IP 地址与 Pod 相关联。
  - 示例：假设一个具有 2 个 ENI 的实例，每个 ENI 支持 5 个 IP 地址。WARM_ENI_TARGET 设置为 1。如果恰好有 5 个 IP 地址与该实例关联，则 CNI 会保留 2 个 ENI 连接到该实例。第一个弹性网卡正在使用中，并且使用了该弹性网卡的所有 5 个可能的 IP 地址。第二个 ENI 是 “热的”，所有 5 个 IP 地址都在池中。如果在实例上启动另一个 Pod，则需要第 6 个 IP 地址。CNI 将为第 6 个 Pod 分配来自第二个 ENI 的 IP 地址和池中的 5 个 IP 地址。第二个 ENI 现已在使用中，不再处于 “热” 状态。CNI 将分配第三个 ENI 来维持至少 1 个热弹性网卡。

> **Note**
>
> 温弹性网卡仍会消耗来自您的 VPC 的 CIDR 的 IP 地址。IP 地址在与工作负载（例如 Pod）关联之前一直是 “未使用” 或 “热的”。

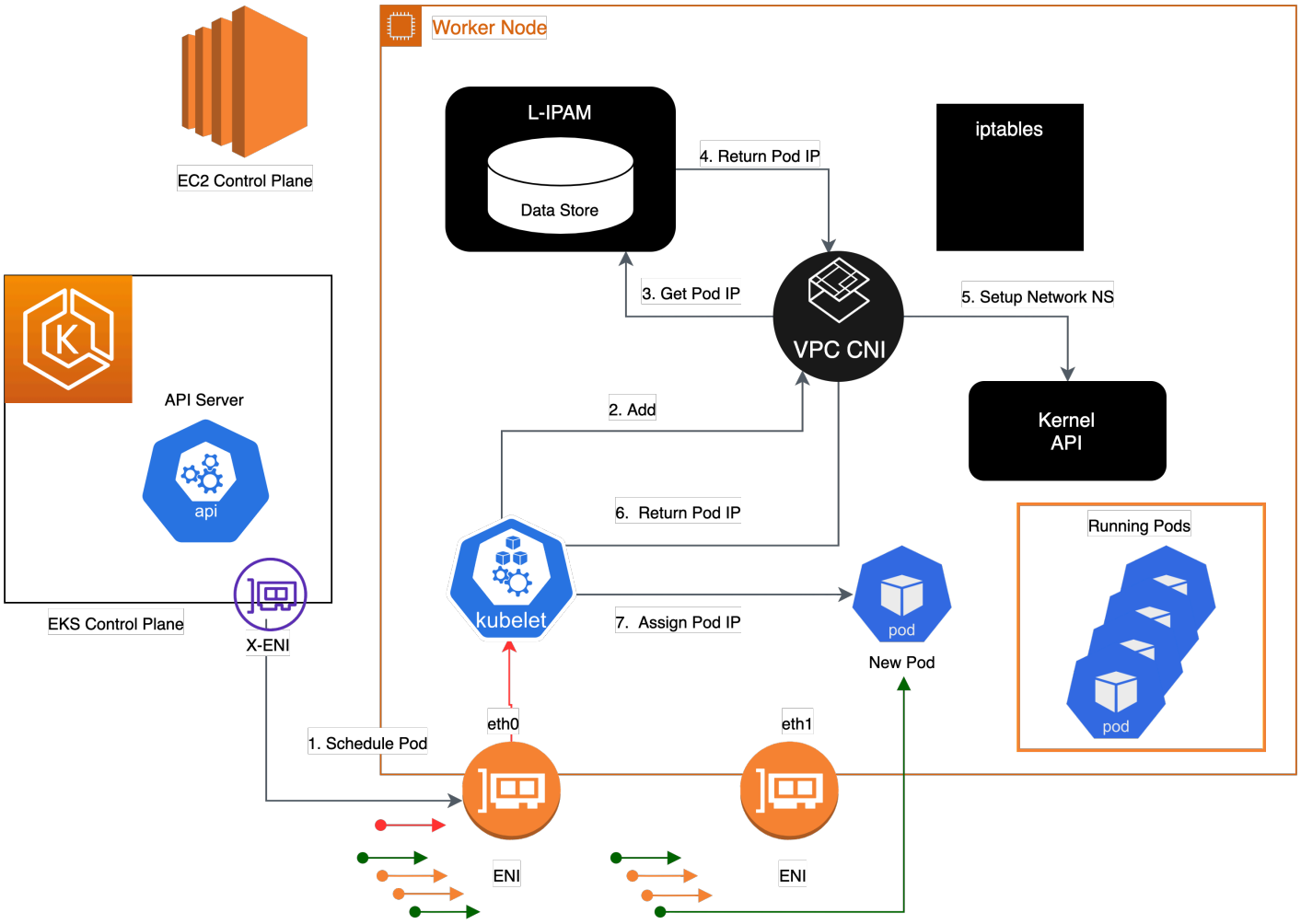
- WARM_IP_TARGET，整数，大于 0 的值表示已启用要求
  - 要维护的温暖 IP 地址的数量。主动连接的 ENI 上有热 IP，但尚未分配给 Pod。换句话说，可用热 IP 的数量是在不需要额外的 ENI 的情况下可以分配给 Pod 的 IP 数量。
- 示例：假设一个具有 1 个 ENI 的实例，每个 ENI 支持 20 个 IP 地址。WARM_IP_TARGET 设置为 5。WARM_ENI_TARGET 设置为 0。在需要第 16 个 IP 地址之前，只会附加 1 个 ENI。然后，CNI 将连接第二个 ENI，占用子网 CIDR 中的 20 个可能的地址。
- MINIMUM_IP_TARGET，整数，大于 0 的值表示已启用要求
  - 随时要分配的最少 IP 地址数。这通常用于在实例启动时预加载多个 ENI 的分配。
  - 示例：考虑一个新启动的实例。它有 1 个弹性网卡，每个弹性网卡支持 10 个 IP 地址。MINIMUM_IP_TARGET 设置为 100。ENI 会立即再连接 9 个 ENI，总共有 100 个地址。无论任何 WARM_IP_TARGET 或 WARM_ENI_TARGET 值如何，都会发生这种情况。

该项目包括子网计算器 Excel 文档。本计算器文档模拟了不同的 ENI 配置选项（例如WARM_IP_TARGET和WARM_ENI_TARGET）下指定工作负载的 IP 地址消耗情况。

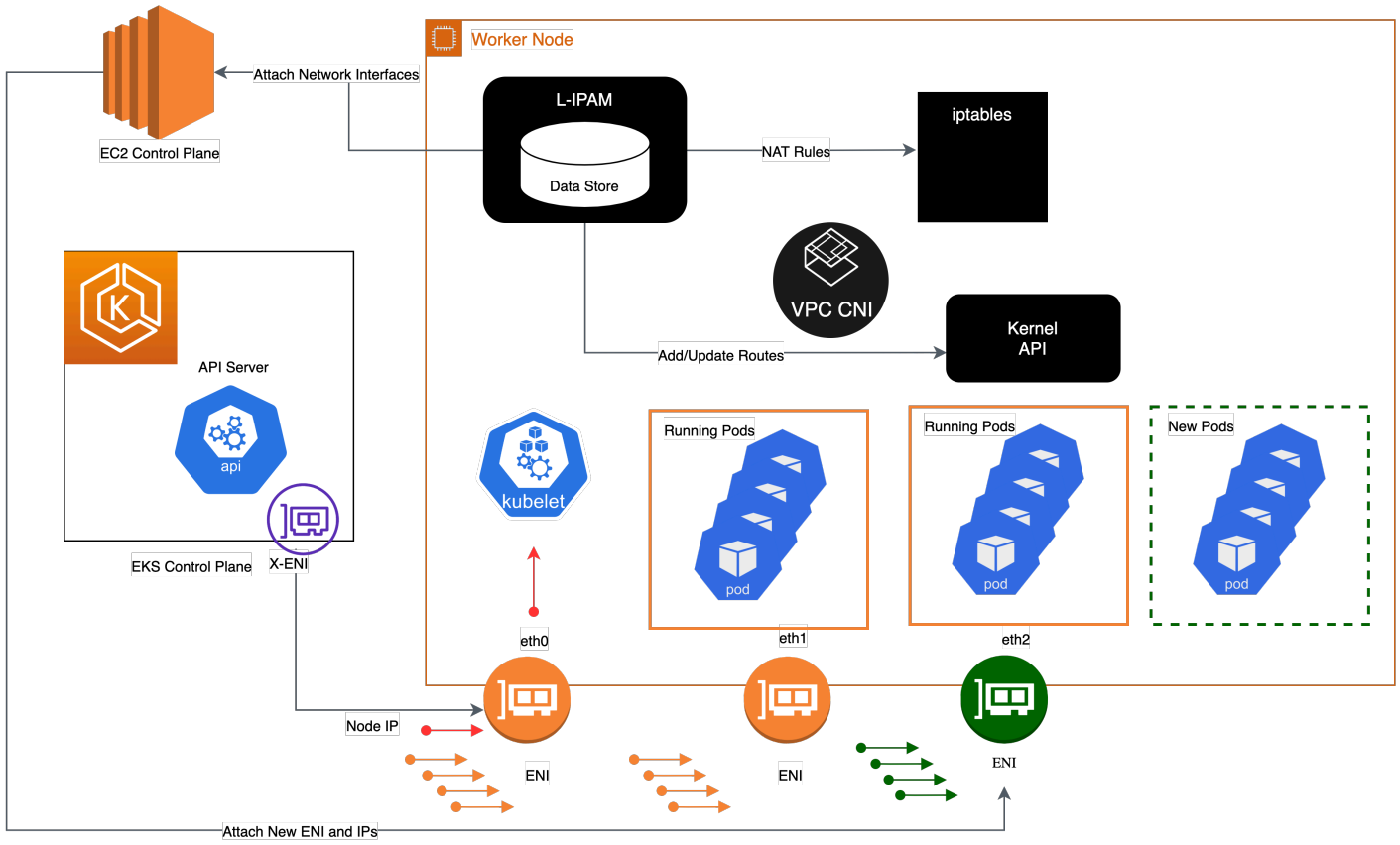



当 Kubelet 收到添加 Pod 请求时，CNI 二进制文件会向 ipamd 查询可用的 IP 地址，然后 ipamd 将其提供给 Pod。CNI 二进制文件连接主机和 Pod 网络。

默认情况下，部署在节点上的 Pod 会被分配到与主 ENI 相同的安全组。或者，Pod 可以配置不同的安全组。




在 IP 地址池耗尽时，该插件会自动附加另一个弹性网络接口到实例，并将另外一组辅助 IP 地址分配到接口。此过程将继续，直到节点不再支持额外的弹性网络接口。




删除 Pod 后，VPC CNI 会将该 Pod 的 IP 地址放在 30 秒的冷却缓存中。冷却缓存中的 IP 不会分配给新 Pod。冷静期结束后，VPC CNI 会将 Pod IP 移回温池。冷却期可防止过早回收 Pod IP 地址，并允许所有集群节点上的 kube-proxy 完成 iptables 规则的更新。当 IP 或 ENI 的数量超过温池设置的数量时，ipamd 插件会将 IP 和 ENI 返回到 VPC。

如上文在辅助 IP 模式中所述，每个 Pod 都会从连接到实例的其中一个 ENI 接收一个辅助私有 IP 地址。由于每个 Pod 都使用一个 IP 地址，因此您可以在特定 EC2 实例上运行的 Pod 数量取决于可以连接到该实例的数量 ENI 以及它支持的 IP 地址数量。VPC CNI 会检查限制文件，以了解每种类型的实例允许使用多少个 ENI 和 IP 地址。

您可以使用以下公式来确定可以在节点上部署的最大 Pod 数量。

```
(Number of network interfaces for the instance type * (the number of IP addresses per
 network interface - 1)) + 2
```

+2 表示需要主机联网的 Pod，例如 kube-proxy 和 VPC CNI。Amazon EKS 要求在每个节点上运行 kube-proxy 和 VPC CNI，这些要求已计入最大容量值中。如果您想运行其他主机网络 Pod，请考虑更新 max-pods 值。您可以在启动模板中指定`--kubelet-extra-args "–max-pods=110"`为用户数据。

例如，在拥有 3 个 c5.large 节点（3 个 ENI 且每个 ENI 最多 10 个 IP）的集群上，当集群启动并拥有 2 个 CoreDNS 容器时，CNI 将消耗 50 个 IP 地址并在温池中保留 43 个 IP。在部署应用程序时，使用热池可以更快地启动 Pod。

节点 1（带有 CoreDNS 容器）：2 个 ENI，分配了 20 个 IP

节点 2（带有 CoreDNS 容器）：2 个 ENI，分配了 20 个 IP

节点 3（没有 Pod）：1 个 ENI。分配了 10 个 IP。

对于节点 1 和节点 2（配置相同）：

- 2 个 ENI × 每个 ENI 10 个 IP = 总共 20 个 IP
- 减去 2 个主 IP（每个 ENI 一个）= 18 个 IP
- 减去 CoreDNS Pod 的 1 个 IP = 17 个可用 IP
- 因此，这些节点中的每个节点在温池中都有 17 个 IP

对于节点 3：

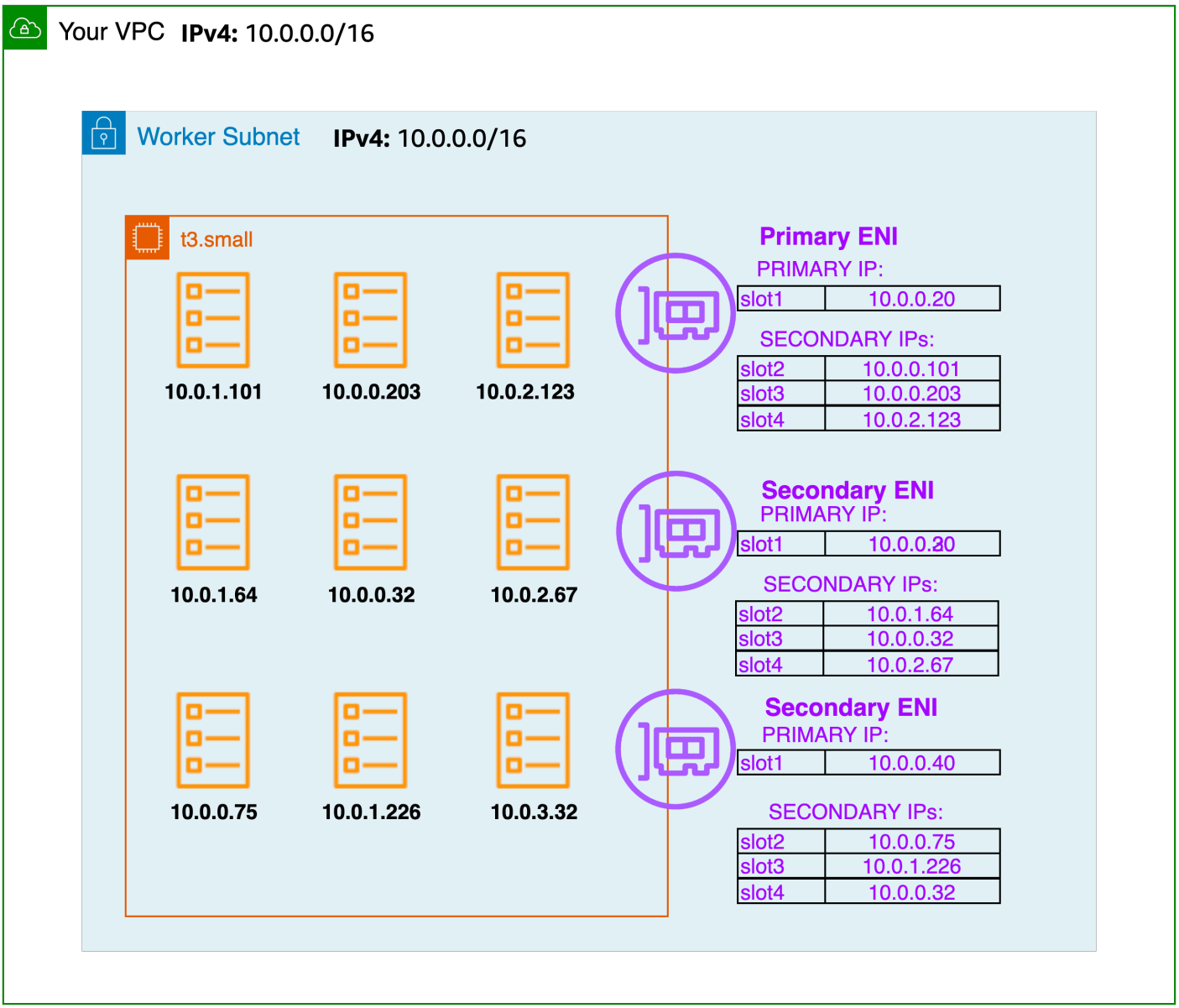
- 1 个 ENI × 10 个 IP = 总共 10 个 IP
- 减去 1 个主 IP = 温池中有 9 个可用的 IP

温池计算总数：-17（节点 1）+ 17（节点 2）+ 9（节点 3）= 43 个 IP

请记住，基础架构 pod 通常作为守护程序集运行，每个容器都会占最大容器数量。这些可能包括：

- CoreDNS
- 亚马逊弹性 LoadBalancer
- 用于指标服务器的操作 pod

我们建议您通过组合这些 Pod 的容量来规划您的基础架构。有关每种实例类型支持的最大 Pod 数的列表，请参阅上的 eni-max-pods.txt GitHub。




# 建议

## 使用自动模式部署 EKS 集群

当您使用 EKS 自动模式创建集群时，AWS 会管理您的集群的 VPC 容器网络接口 (CNI) 配置。使用 Amazon EKS 自动模式时，您无需安装或升级联网附加组件。但是，请确保您的工作负载与托管 VPC CNI 配置兼容。

## 部署 VPC CNI 托管 Add-On

当您配置集群时，Amazon EKS 会自动安装 VPC CNI。不过，Amazon EKS 支持托管插件，使集群能够与计算、存储和网络等底层 AWS 资源进行交互。我们强烈建议您使用托管插件（包括 VPC CNI）部署集群。

Amazon EKS 托管插件为亚马逊 EKS 集群提供 VPC CNI 安装和管理。Amazon EKS 插件包含最新的安全补丁和错误修复，并经过 AWS 验证，可与 Amazon EKS 配合使用。VPC CNI 插件使您能够持续确保 Amazon EKS 集群的安全和稳定，并减少安装、配置和更新插件所需的工作量。此外，还可以通过 Amazon EKS API、AWS 管理控制台、AWS CLI 和 eksctl 添加、更新或删除托管插件。

您可以使用带有`kubectl get`命令的`--show-managed-fields`标志查找 VPC CNI 的托管字段。

```
kubectl get daemonset aws-node --show-managed-fields -n kube-system -o yaml
```

托管插件通过每 15 分钟自动覆盖配置来防止配置偏移。这意味着，在插件创建后通过 Kubernetes API 对托管插件所做的任何更改都将被自动漂移防护流程覆盖，并在插件更新过程中设置为默认值。

由 EKS 管理的字段列在 managedFields 下，管理器名为 EKS。EKS 管理的字段包括服务帐户、图像、图像网址、存活探测、就绪探测、标签、卷和卷挂载。

> **Note**
>
> 最常用的字段，例如 WARM_ENI_TARGET、WARM_IP_TARGET 和 MINIMUM_IP_TARGET，不受管理，也不会被协调。这些字段的更改将在插件更新后保留。

我们建议在更新生产集群之前，针对特定配置在非生产集群中测试插件行为。此外，请按照 EKS 用户指南中的步骤进行附加配置。

迁移到托管 Add-On

您将管理自管理 VPC CNI 的版本兼容性并更新安全补丁。要更新自行管理的插件，您必须使用 Kubernetes API 和 EKS 用户指南中概述的说明。我们建议将现有 EKS 集群迁移到托管插件，并强烈建议在迁移之前创建当前 CNI 设置的备份。要配置托管插件，您可以使用 Amazon EKS API、AWS 管理控制台或 AWS 命令行界面。

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

如果该字段被列为使用默认设置托管字段，则 Amazon EKS 将替换 CNI 配置设置。我们谨慎行事，不要修改托管字段。该插件无法协调配置字段，例如温暖环境变量和 CNI 模式。当您迁移到托管 CNI 时，Pod 和应用程序将继续运行。

更新前备份 CNI 设置

VPC CNI 在客户数据平面（节点）上运行，因此，在新版本发布时或将集群更新到新的 Kubernetes 次要版本之后，Amazon EKS 不会自动更新插件（托管和自行管理）。要更新现有集群的插件，您必须通过 update-addon API 触发更新，或者单击 EKS 控制台中的立即更新链接以获取插件。如果您已经部署了自行管理的插件，请按照更新自管 VPC CN I 插件中提及的步骤进行操作。

我们强烈建议您一次更新一个次要版本。例如，如果您的当前的次要版本为 `1.9`，并且您想要更新到 `1.11`，则应首先更新到最新的补丁版本 `1.10`，再更新到最新的补丁版本 `1.11`。

在更新 Amazon VPC CNI 之前，请对 aws-node 守护程序集进行检查。备份现有设置。如果使用托管插件，请确认您尚未更新 Amazon EKS 可能覆盖的任何设置。我们建议在您的自动化工作流程中使用更新后挂钩，或者在插件更新后手动应用步骤。

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

对于自行管理的插件，请将备份与 `releases` on 进行比较 GitHub 以查看可用版本并熟悉要更新到的版本中的更改。我们建议使用 Helm 来管理自我管理的插件，并利用值文件来应用设置。任何涉及 Daemonset 删除的更新操作都将导致应用程序停机，必须避免。

## 了解安全背景

我们强烈建议您了解为高效管理 VPC CNI 而配置的安全上下文。亚马逊 VPC CNI 有两个组成部分 CNI 二进制文件和 ipamd（aws-node）Daemonset。CNI 作为二进制文件在节点上运行，可以访问节点根文件系统，还具有特权访问权限，因为它在节点级别处理 iptables。当添加或删除 Pod 时，kubelet 会调用 CNI 二进制文件。

aws-node Daemonset 是一个长期运行的进程，负责节点级别的 IP 地址管理。aws-node 在`hostNetwork`模式下运行，允许访问环回设备以及同一节点上其他 Pod 的网络活动。aws-node 初始化容器在特权模式下运行并挂载 CRI 套接字，允许 Daemonset 监控节点上运行的 Pod 的 IP 使用情况。Amazon EKS 正在努力删除 aws-node 初始化容器的特权要求。此外，aws-node 需要更新 NAT 条目并加载 iptables 模块，因此以 NET_ADMIN 权限运行。

Amazon EKS 建议部署由 aws-node 清单定义的安全策略，用于 Pod 和网络设置的 IP 管理。请考虑更新到最新版本的 VPC CNI。此外，如果您有特定的安全要求，请考虑提出GitHub 问题。

## 为 CNI 使用单独的 IAM 角色

AWS VPC CNI 需要 AWS Identity and Access Management (IAM) 权限。需要先设置 CNI 策略，然后才能使用 IAM 角色。您可以使用 AmazonEKS_CNI_Policy，这是一项适用于 IPv4 集群的 AWS 托管策略。Amazoneks CNI 托管策略仅具有 IPv4 集群的权限。您必须使用此处列出的权限为 IPv6 集群创建单独的 IAM 策略。

默认情况下，VPC CNI 继承 A mazon EKS 节点 IAM 角色（包括托管节点组和自管理节点组）。

强烈建议使用适用于 Amazon VPC CNI 的相关策略配置单独的 IAM 角色。否则，Amazon VPC CNI 的 pod 将获得分配给节点 IAM 角色的权限，并可以访问分配给该节点的实例配置文件。

VPC CNI 插件可创建并配置一个名为 aws-node 的服务账户。默认情况下，服务账户绑定到附有 Amazon EKS CNI 政策的 Amazon EKS 节点 IAM 角色。要使用单独的 IAM 角色，我们建议您创建一个附有 Amazon EKS CNI 政策的新服务账户。要使用新的服务帐号，必须重新部署 CNI 容器。创建新集群时，请考虑`--service-account-role-arn`为 VPC CNI 托管插件指定。确保从亚马逊 EKS 节点角色中移除适用于 IPv4 和 IPv6 的 Amazon EKS CNI 政策。

建议您屏蔽访问实例元数据，以最大限度地减少安全漏洞的爆炸半径。

## 处理 Liveness/Readiness 探测失败

我们建议增加 EKS 1.20 及更高版本集群的活动和就绪探测超时值（默认`timeoutSeconds: 10`），以防止探测失败导致应用程序的 Pod 卡在 ContainerCreating 状态。在数据密集型和批处理集群中已出现此问题。CPU 使用率过高会导致 aws-node 探测运行状况故障，从而导致 Pod CPU 请求未得到满足。除了修改探测超时外，还要确保正确配置 aws-node 的 CPU 资源请求（默认`CPU: 25m`）。除非您的节点出现问题，否则我们不建议更新设置。

我们强烈建议您在接触 Amazon EKS 支持时在节点`bash /opt/cni/bin/aws-cni-support.sh`上运行 sudo。该脚本将帮助评估节点上的 kubelet 日志和内存利用率。请考虑在 Amazon EKS 工作节点上安装 SSM 代理来运行脚本。

## 在非 EKS 优化的 AMI 实例上配置 IPTables 转发策略

如果你使用的是自定义 AMI，请务必在 kub elet.service 下将 iptables 转发策略设置为 “接受”。许多系统将 iptables 转发策略设置为 DROP。您可以使用 HashiCorp Packer 构建自定义 AMI，并使用来自 AWS 上的 Amazon EKS GitHub AMI 存储库的资源和配置脚本构建规范。您可以更新 kubelet.service 并按照此处指定的说明创建自定义 AMI。

### 定期升级 CNI 版本

VPC CNI 向后兼容。最新版本适用于 Amazon EKS 支持的所有 Kubernetes 版本。此外，VPC CNI 作为 EKS 附加组件提供（参见上文的 “部署 VPC CNI 托管 Add-On”）。虽然 EKS 插件会协调插件的升级，但它不会自动升级像 CNI 这样的插件，因为它们是在数据平面上运行的。在托管和自我管理的工作节点升级之后，您负责升级 VPC CNI 插件。

## 优化 IP 地址利用率

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

得益于应用程序现代化，容器化环境的规模正在快速增长。这意味着部署的工作节点和 Pod 越来越多。

Amazon VPC CNI 插件为每个容器分配一个来自 VPC 的 CIDR 的 IP 地址。这种方法通过 VPC 流日志和其他监控解决方案等工具提供 Pod 地址的全面可见性。根据您的工作负载类型，这可能会导致 Pod 消耗大量的 IP 地址。

在设计 AWS 网络架构时，在 VPC 和节点级别优化 Amazon EKS IP 使用非常重要。这将帮助您缓解 IP 耗尽问题并增加每个节点的 pod 密度。

在本节中，我们将讨论可以帮助您实现这些目标的技术。

### 优化节点级 IP 消耗

前缀委派是亚马逊虚拟私有云（亚马逊 VPC）的一项功能，它允许您为亚马逊弹性计算云 (Amazon EC2) 实例分配 IPv4 或 IPv6 前缀。它增加了每个网络接口 (ENI) 的 IP 地址，从而提高了每个节点的 pod 密度并提高了计算效率。自定义网络还支持前缀委托。

有关详细信息，请参阅 Linux 节点的前缀委派和使用 Windows 节点的前缀委派部分。

### 减轻 IP 耗尽的风险

为防止您的集群消耗所有可用的 IP 地址，我们强烈建议您在调整您的 VPC 和子网时考虑增长。

采用 IPv6 是从一开始就避免这些问题的好方法。但是，对于可扩展性需求超过初始规划且无法采用 IPv6 的组织，建议改进 VPC 设计是应对 IP 地址耗尽的应对措施。Amazon EKS 客户最常用的技术是

向 VPC 添加不可路由的辅助 CIDR，并将 VPC CNI 配置为在为 Pod 分配 IP 地址时使用额外的 IP 空间。这通常被称为自定义网络。

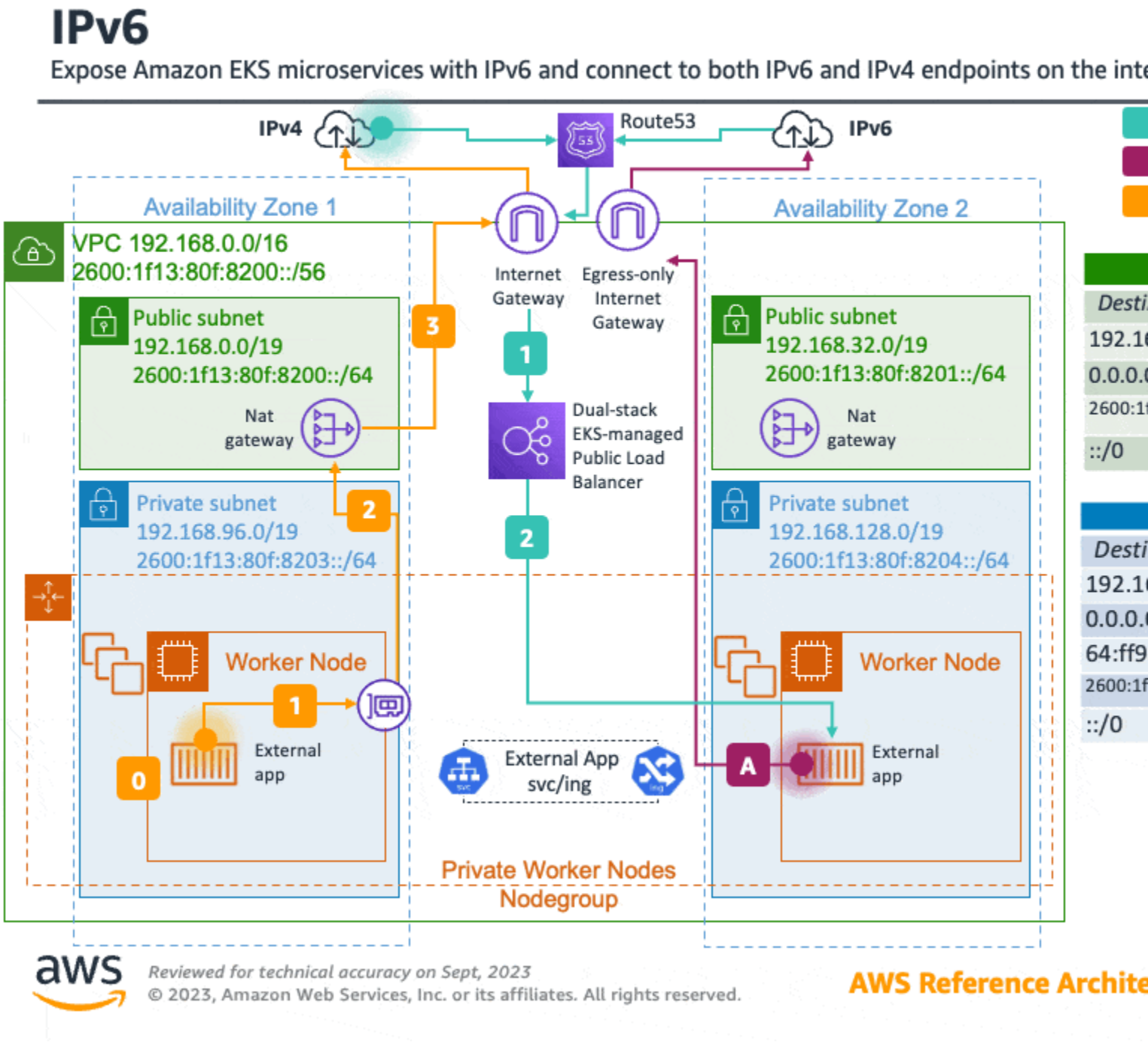
我们将介绍您可以使用 Amazon VPC CNI 的哪些变量来优化分配给节点的 IP 热池。在本节结束时，我们将介绍一些其他架构模式，这些模式不是 Amazon EKS 固有的，但可以帮助缓解 IP 耗尽。

## 使用 IPv6（推荐）

采用 IPv6 是解决 RFC1918 限制的最简单方法；我们强烈建议您在选择网络架构时考虑将 IPv6 作为首选。IPv6 提供的总 IP 地址空间要大得多，集群管理员可以专注于迁移和扩展应用程序，而不必花精力绕过 IPv4 限制。

亚马逊 EKS 集群同时支持 IPv4 和 IPv6。默认情况下，EKS 集群使用 IPv4 地址空间。在创建集群时指定基于 IPv6 的地址空间将启用 IPv6。在 IPv6 EKS 集群中，容器和服务接收 IPv6 地址，同时保持传统 IPv4 终端节点连接到在 IPv6 集群上运行的服务的能力，反之亦然。集群内所有 pod 到 Pod 的通信始终通过 IPv6 进行。在 VPC (/56) 中，IPv6 子网的 IPv6 CIDR 块大小固定为 /64。这提供了 2^64（大约 18 万亿）的 IPv6 地址，允许您在 EKS 上扩展部署。

有关详细信息，请参阅 “运行 IPv6 EKS 集群” 部分；要获得实践经验，请参阅 “亲身体验 IPv6” 研讨会的 “了解 Amazon EKS 上的 IPv6” 部分。




## 优化 IPv4 集群中的 IP 消耗

本节专门针对正在运行传统应用程序但尚未准备好迁移到 IPv6 的客户。 and/or 虽然我们鼓励所有组织尽快迁移到 IPv6，但我们认识到，有些组织可能仍需要考虑其他方法来使用 IPv4 扩展其容器工作负载。为此，我们还将引导您了解架构模式，以优化 Amazon EKS 集群的 IPv4 (RFC1918) 地址空间消耗。

### 为增长做好计划

作为防止 IP 耗尽的第一道防线，我们强烈建议您在考虑增长的情况下调整 IPv4 VPC 和子网的大小，以防止您的集群消耗所有可用 IP 地址。如果子网没有足够的可用的 IP 地址，您将无法创建新的 Pod 或节点。

在构建 VPC 和子网之前，建议从所需的工作负载规模向后移动。例如，使用 eksctl（用于在 EKS 上创建和管理集群的简单 CLI 工具）构建集群时，默认情况下会创建 19 个子网。/19 的网络掩码适用于允许分配超过 8000 个地址的大多数工作负载类型。

> **Important**
>
> 在调整 VPC 和子网的大小时，可能会有许多元素（Pod 和节点除外）会消耗 IP 地址，例如负载均衡器、RDS 数据库和其他 vpc 内服务。

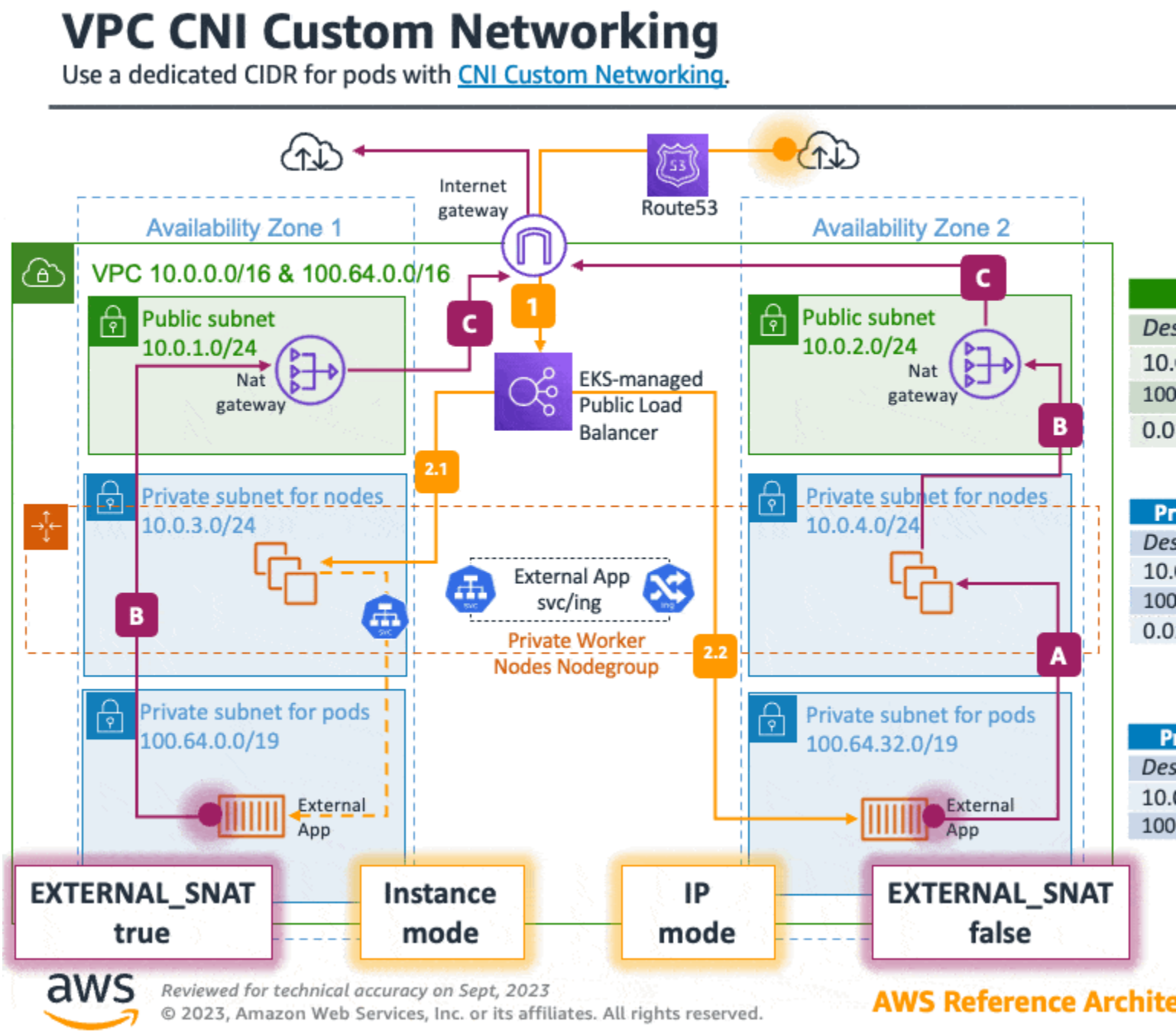
此外，Amazon EKS 可以创建多达 4 个弹性网络接口 (X-ENI)，这些接口是允许与控制平面通信所必需的（更多信息请点击此处）。在集群升级期间，Amazon EKS 会创建新的集群， X-ENIs 并在升级成功后删除旧集群。因此，对于与 EKS 集群关联的子网，我们建议网络掩码至少为 /28（16 个 IP 地址）。

您可以使用 EKS 子网计算器电子表格样本来规划您的网络。该电子表格根据工作负载和 VPC ENI 配置计算 IP 使用情况。将 IP 使用情况与 IPv4 子网进行比较，以确定配置和子网大小是否足以应付您的工作负载。请记住，如果您的 VPC 中的子网用完了可用 IP 地址，我们建议使用 VPC 的原始 CIDR 块创建一个新子网。请注意，现在 Amazon EKS 允许修改集群子网和安全组。

### 自定义网络

如果您即将耗尽 RFC1918 IP 空间，则可以使用自定义网络模式通过在专用的额外子网内调度 Pod 来节省可路由的 IP。虽然自定义网络将接受辅助 CIDR 范围的有效 VPC 范围，但我们建议您使用`100.64.0.0/10`共享地址空间 (RFC 6598) 中的 CIDR，因为与 RFC1918 范围相比，这些地址在企业环境中使用的可能性要小。例如，您可以`100.64.0.0/16`将其用作您的 VPC 的辅助 CIDR。有关允许的辅助 CIDR 范围，请参阅 IPv4 CIDR 区块关联限制。

有关详细信息，请参阅“自定义网络”的专用部分。



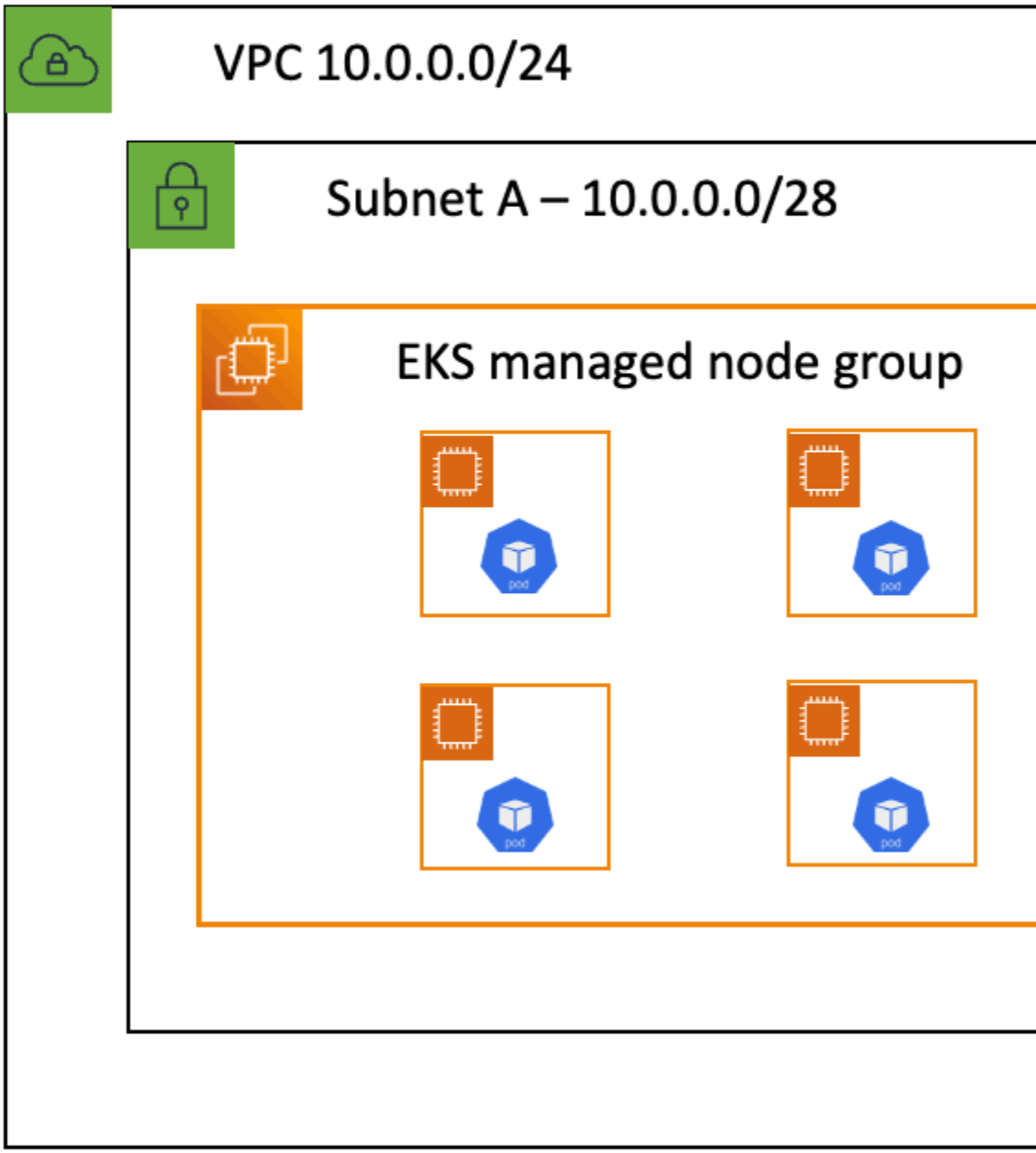

增强的子网发现

增强型子网发现通过标记新子网，使 Amazon VPC CNI 可以发现这些子网，从而为 IP 耗尽提供了一种简化的网络配置替代方案。借助增强型子网发现，当前的工作负载可以在相同的子网上继续运行，Amazon Elastic Kubernetes Service (Amazon EKS) 现在可以在新的 “可用子网” 上安排更多容器。

如果您的集群当前子网的 IP 地址已用完，则只需按如下方式向 Amazon EKS 集群添加其他子网即可：

1. 将新的 CIDR 块关联到您的 VPC。
2. 在新的 CIDR 块中创建一个新的子网，并用 “kubernetes” 对其进行标记。 io/role/cni” = “1"。
3. 将 Amazon VPC CNI 插件的 ENABLE_SUBNET_DISCOVERY 配置设置为 “true”（自版本 1.18.0 起默认）。

在您的 VPC 和 Amazon EKS 集群上启用增强型子网发现后，新的弹性网络接口 (ENI) 将连接到您的 Amazon EKS 节点，如下图所示：


VPC 10.0.0.0/24
Subnet A – 10.0.0.0/28
EKS managed node group
pod
pod
pod
pod

有关更多信息，请参阅 AWS 容器博客上的 Amazon VPC CNI 推出增强型子网发现。

优化 IP 的温池

在默认配置下，VPC CNI 会将整个 ENI（和相关的 IP）保留在温池中。这可能会消耗大量 IP，尤其是在较大的实例类型上。

如果您的集群子网可用的 IP 地址数量有限，请仔细检查以下 VPC CNI 配置环境变量：

- `WARM_IP_TARGET`
- `MINIMUM_IP_TARGET`
- `WARM_ENI_TARGET`

您可以将的值配置`MINIMUM_IP_TARGET`为与您期望在节点上运行的 Pod 数量非常匹配。这样做可以确保在创建 Pod 时，CNI 无需调用 EC2 API 即可从温池中分配 IP 地址。

请注意，将值设置`WARM_IP_TARGET`得太低会导致对 EC2 API 的额外调用，这可能会导致请求受到限制。对于大型集群，使用和`MINIMUM_IP_TARGET`以避免限制请求。

要配置这些选项，您可以下载`aws-k8s-cni.yaml`清单并设置环境变量。在撰写本文时，最新版本位于此处。检查配置值的版本是否与已安装的 VPC CNI 版本相匹配。

> **Warning**
>
> 更新 CNI 时，这些设置将重置为默认值。在更新 CNI 之前，请对其进行备份。查看配置设置以确定更新成功后是否需要重新应用它们。

您可以即时调整现有应用程序的 CNI 参数，而无需停机，但您应该选择能够满足您的可扩展性需求的值。例如，如果您使用的是批处理工作负载，我们建议您更新默认工作负载`WARM_ENI_TARGET`以满足 Pod 规模的需求。设置`WARM_ENI_TARGET`为较高的值将始终保持运行大批量工作负载所需的热 IP 池，从而避免数据处理延迟。

> **Warning**
>
> 改进 VPC 设计是应对 IP 地址耗尽的建议措施。可以考虑像 IPv6 和辅助 CIDR 这样的解决方案。在排除其他选项之后，调整这些值以最大限度地减少热 IP 的数量应该是一种临时解决方案。错误配置这些值可能会干扰群集操作。在对生产系统进行任何更改之前，请务必查看此页面上的注意事项。

监控 IP 地址清单

除了上述解决方案外，了解 IP 利用率也很重要。您可以使用 CNI 指标助手监控子网的 IP 地址清单。一些可用的指标是：

- 集群可以支持的最大 ENI 数量
- 已分配的弹性网卡数量
- 当前分配给 Pod 的 IP 地址数量
- 可用 IP 地址的总数和最大数量

您还可以设置CloudWatch 警报，以便在子网的 IP 地址用完时收到通知。

> **Warning**
>
> 确保 VPC CNI 的`DISABLE_METRICS`变量设置为 false。

进一步的考虑

还有其他不是 Amazon EKS 固有的架构模式可以帮助解决 IP 耗尽问题。例如，您可以优化 VPC 间的通信或跨多个账户共享 VPC 以限制 IPv4 地址分配。

在此处了解有关这些模式的更多信息：

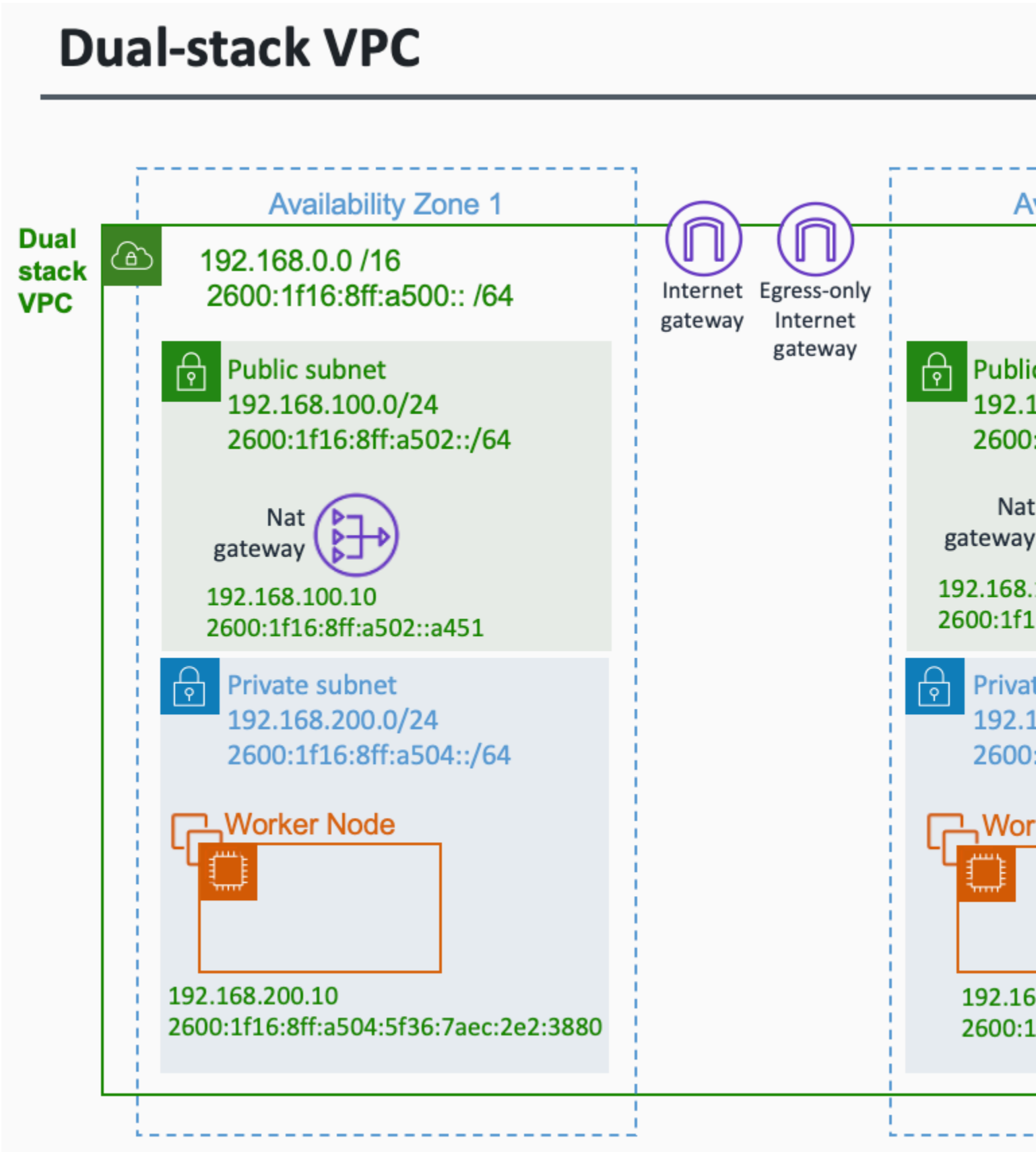
- 设计超大规模的 Amazon VPC 网络，
- 使用 Amazon VPC Lattice 建立安全的多账户多 VPC 连接。

# 运行 IPv6 EKS 集群

IPv6 模式下的 EKS 解决了大规模 EKS 集群中经常出现的 IPv4 耗尽难题。EKS 对 IPv6 的支持侧重于解决 IPv4 耗尽问题，该问题源于 IPv4 地址空间的有限规模。这是我们的许多客户提出的一个重大问题，与 Kubernetes IPv4/IPv6 的双栈功能截然不同。 EKS/IPv6 还将提供使用 IPv6 CIDR 互连网络边界的灵活性，从而最大限度地减少出现 CIDR 重叠的机会，从而解决双重问题（ ， ）。In-Cluster Cross-Cluster在 IPv6 模式（--ip-family ipv6）下部署 EKS 集群时，该操作是不可逆的。简而言之，EKS IPv6 支持将在集群的整个生命周期内启用。

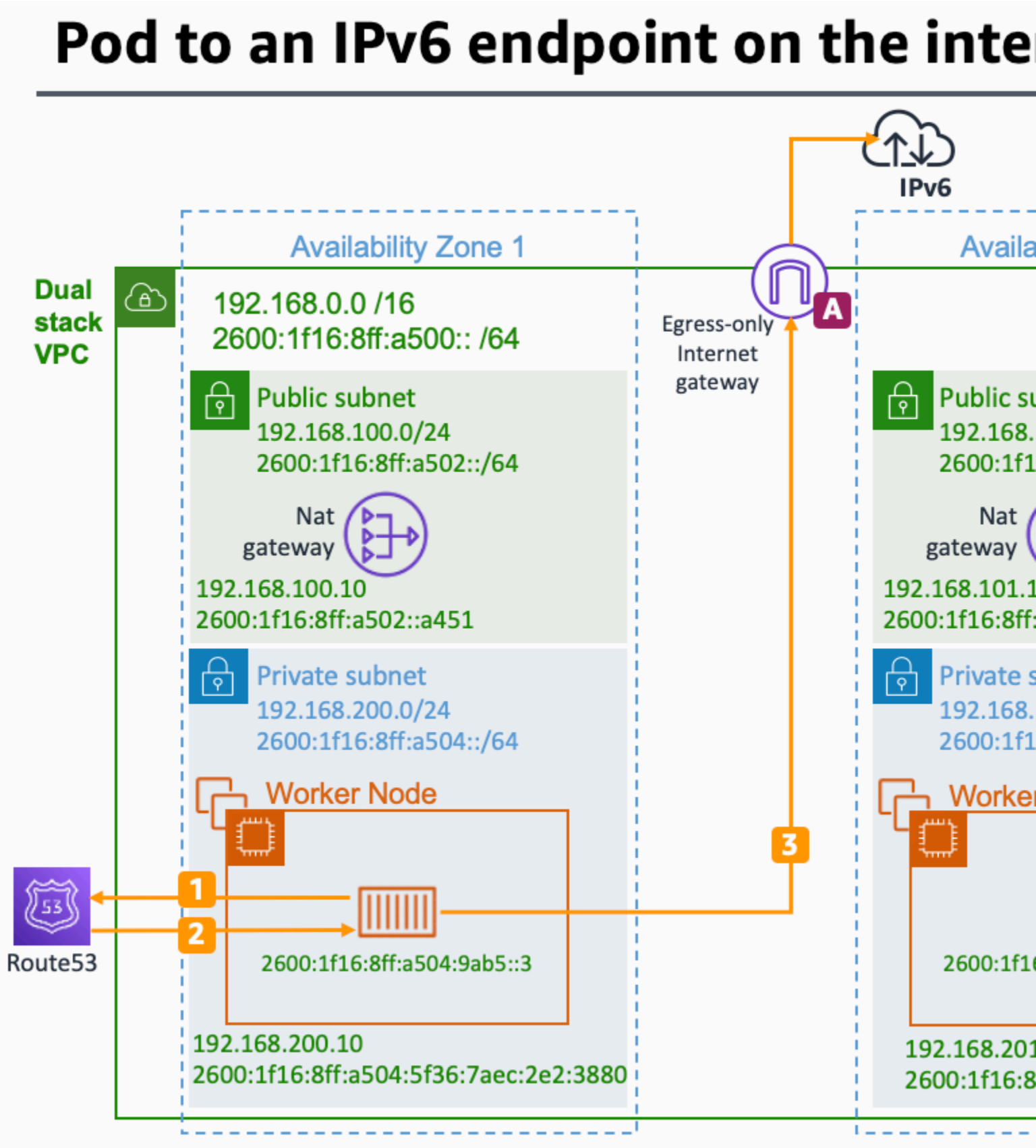
在 IPv6 EKS 集群中，Pod 和服务将接收 IPv6 地址，同时保持与传统 IPv4 终端节点的兼容性。这包括外部 IPv4 端点能够访问集群内服务，以及 Pod 可以访问外部 IPv4 端点。

Amazon EKS IPv6 支持利用了原生 VPC IPv6 功能。每个 VPC 都分配了一个 IPv4 地址前缀（CIDR 块大小可以介于 /16 到 /28 之间）和来自亚马逊 GUA（全球单播地址）的唯一 /56 IPv6 地址前缀（固定）；您可以为 VPC 中的每个子网分配一个 /64 地址前缀。在支持 IPv6 的 VPC 中，路由表、网络访问控制列表、对等互连和 DNS 解析等 IPv4 功能的工作方式相同。然后，VPC 被称为双栈 VPC，按照双栈子网，下图描述了支持基于群集的 IPv4IPv6 VPC 基础模式： EKS/IPv6


Dual-stack VPC
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Egress-only Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

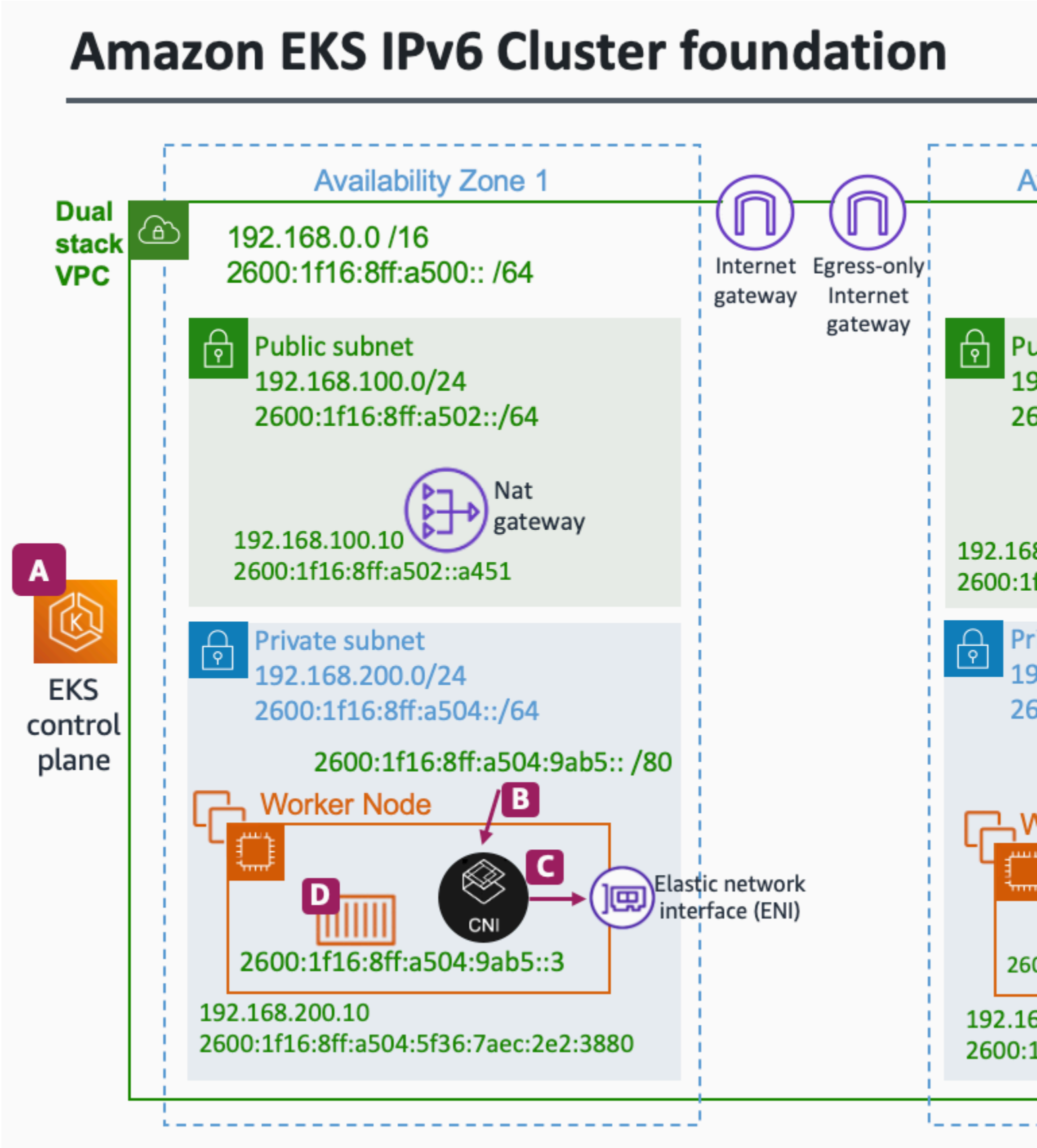
在 IPv6 世界中，每个地址都可以通过互联网进行路由。默认情况下，VPC 会从公共 GUA 范围内分配 IPv6 CIDR。但是，自 2024 年 8 月起，您还可以通过亚马逊 VPC IP 地址管理器 (IPAM) 为 VPC 和子网使用私有 IPv6 寻址。有关更多信息，请参阅此 AWS 网络博客文章和 VPC 文档。

下图描绘了集群内的 Pod IPv6 互联网出口流： EKS/IPv6


Pod to an IPv6 endpoint on the inte
IPv6
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Egress-only Internet gateway
A
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
1
2
3
Route53
2600:1f16:8ff:a504:9ab5::3
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

在 V PC 用户指南中可以找到实施 IPv6 子网的最佳实践。

在 IPv6 EKS 集群中，节点和 Pod 接收公有 IPv6 地址。EKS 根据唯一的本地 IPv6 单播地址 (ULA) 为服务分配 IPv6 地址。与 IPv4 不同，IPv6 集群的 ULA 服务 CIDR 是在集群创建阶段自动分配的，无法指定。下图描绘了 EKS/IPv6 基于集群控制平面数据计划基础模式：


Amazon EKS IPv6 Cluster foundation
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Egress-only Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
A
EKS control plane
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
2600:1f16:8ff:a504:9ab5:: /80
Worker Node
B
C
D
CNI
Elastic network interface (ENI)
2600:1f16:8ff:a504:9ab5::3
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

# 概述

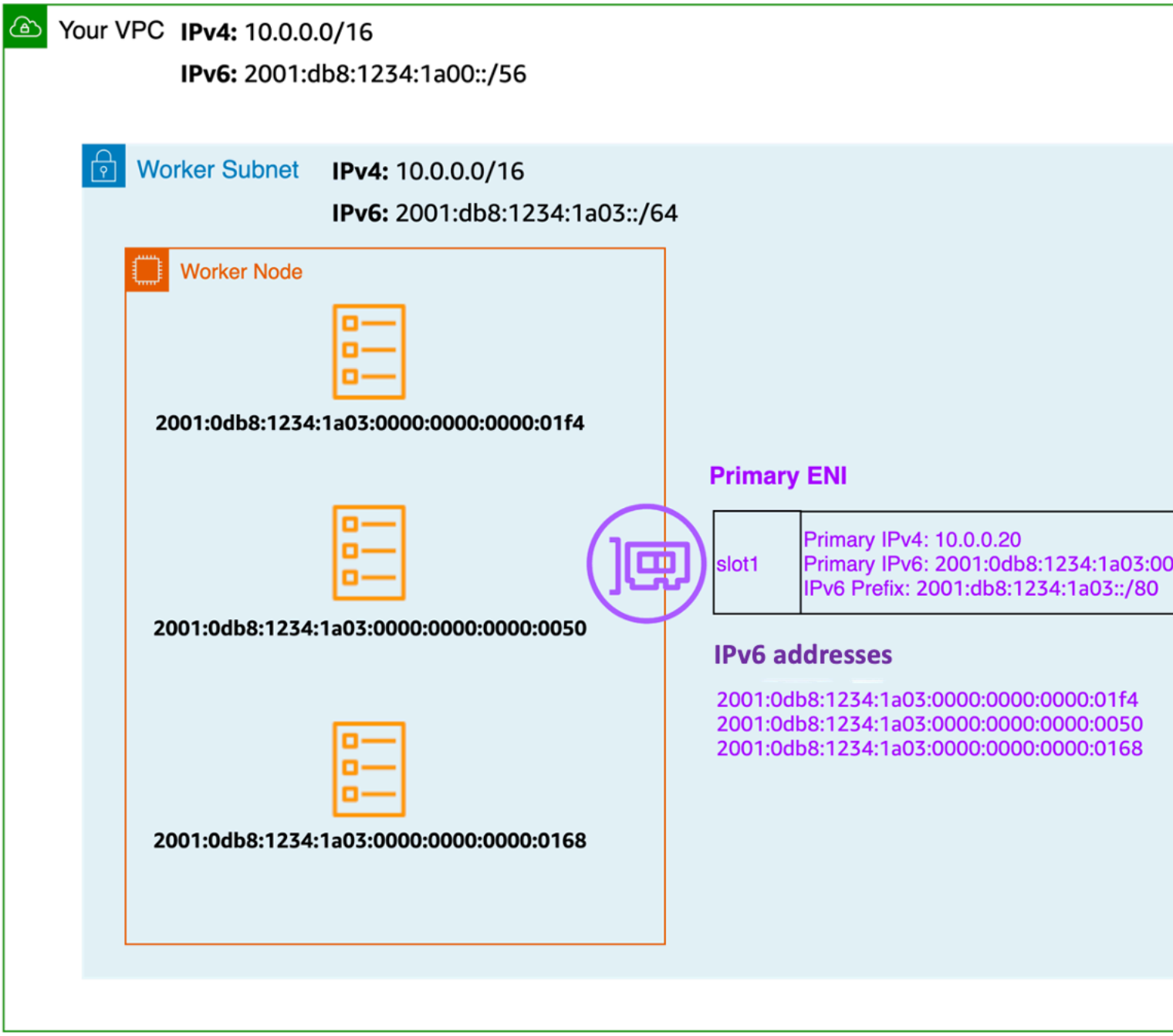
EKS/IPv6 仅在前缀模式（VPC-CNI Plug-in ENI IP 分配模式）中支持。了解有关前缀模式的更多信息。

前缀分配仅适用于 Nitro-based EC2 实例，因此 EKS/IPv6 仅在集群数据平面使用 EC2 Nitro-based 实例时才支持前缀分配。

简而言之，IPv6 前缀为 /80（每个工作节点）将产生大约 10^14 个 IPv6 地址，限制因素将不再是 IP，而是 Pod 密度（资源方面）。

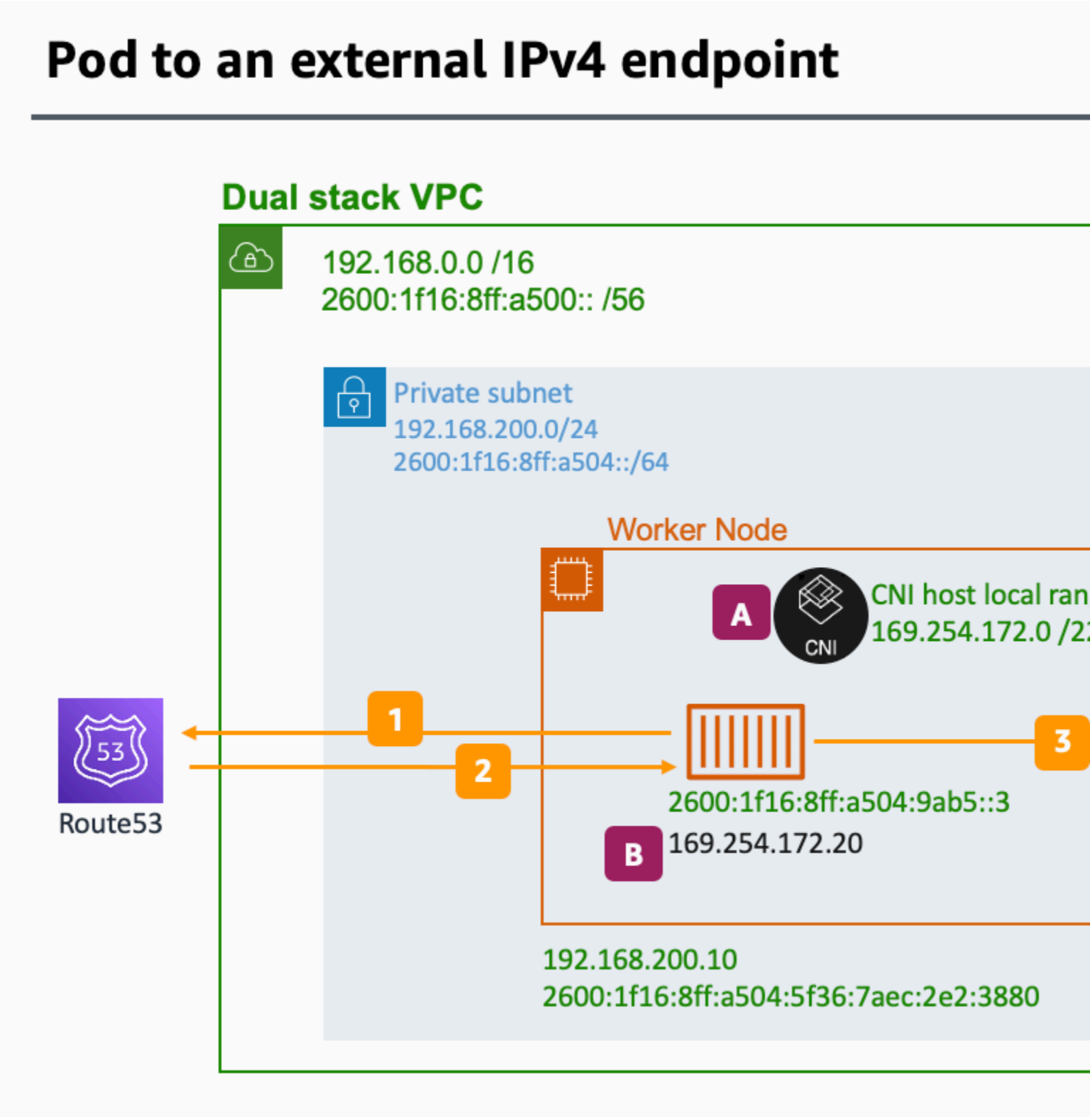
IPv6 前缀分配仅在 EKS 工作节点引导时发生。众所周知，这种行为可以缓解由于旨在及时分配私有 IPv4 地址的 VPC CNI 插件 (ipamd) 生成的 API 调用受限，Pod 流失率高的 EKS/IPv4 集群经常在 Pod 调度中延迟的情况。众所周知， VPC-CNI 插件会不必要地调高高级旋钮 WARM_IP/ENI，MINIMUM_IP。

下图放大了 IPv6 工作节点弹性网络接口 (ENI)：




每个 EKS 工作节点都分配了 IPv4 和 IPv6 地址以及相应的 DNS 条目。对于给定的工作节点，仅使用来自双栈子网的单个 IPv4 地址。EKS 对 IPv6 的支持使您能够通过高度自以为是的仅限出口 IPv4 模式与 IPv4 终端节点（AWS、本地、互联网）进行通信。EKS 实现了一个主机本地 CNI 插件，该插件是 VPC CNI 插件的辅助插件，后者为 Pod 分配和配置 IPv4 地址。CNI 插件为 169.254.172 中的 Pod 配置主机特定的不可路由的 IPv4 地址。 0/22 范围。分配给 Pod 的 IPv4 地址是工作节点所独有的，不会在工作节点之外通告。169.254.172。 0/22 提供多达 1024 个唯一的 IPv4 地址，可以支持大型实例类型。

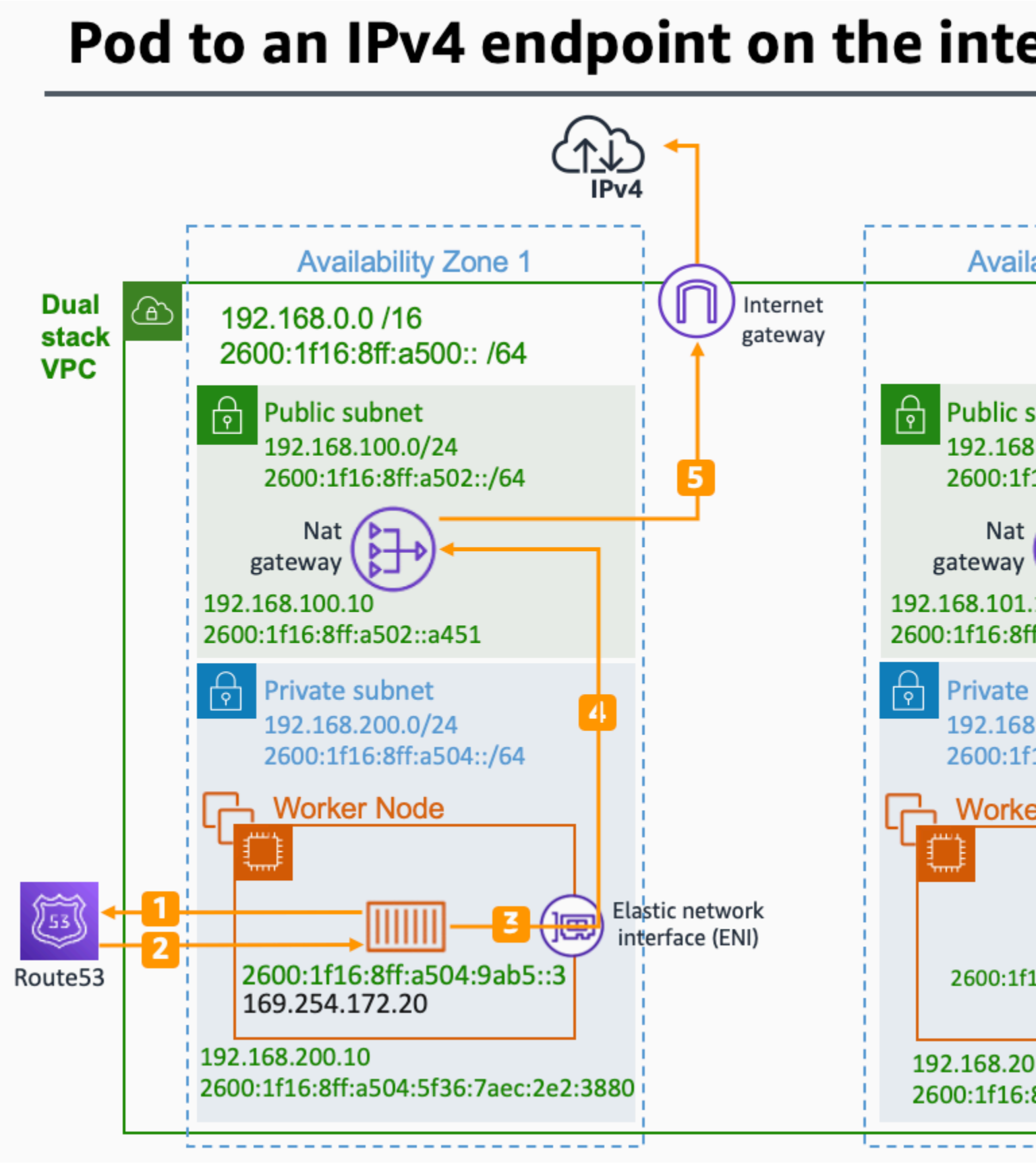
下图描述了 IPv6 Pod 连接到集群边界外（非互联网）的 IPv4 端点的流程：



在上图中，Pod 将对终端节点执行 DNS 查找，在收到 IPv4 “A” 响应后，Pod 的仅限节点的唯一 IPv4 地址通过源网络地址转换 (SNAT) 转换为连接到 EC2 的主网络接口的私有 IPv4 (VPC) 地址。 Worker-node

> **Note**
>
> 上述模式要求在运行 EKS/IPv6 Pod 的子网上禁用 DNS64。启用 DNS64 后，DNS 解析器会返回 IPv4-only 终端节点的合成 IPv6 地址以及 IPv4 地址。因此，流量会通过 NAT 网关（如果包含在架构中）的 NAT64 功能进行路由，而不是像上面的模式所示停留在 VPC 内。这可能会导致意外的 NAT 网关使用量和相关成本。

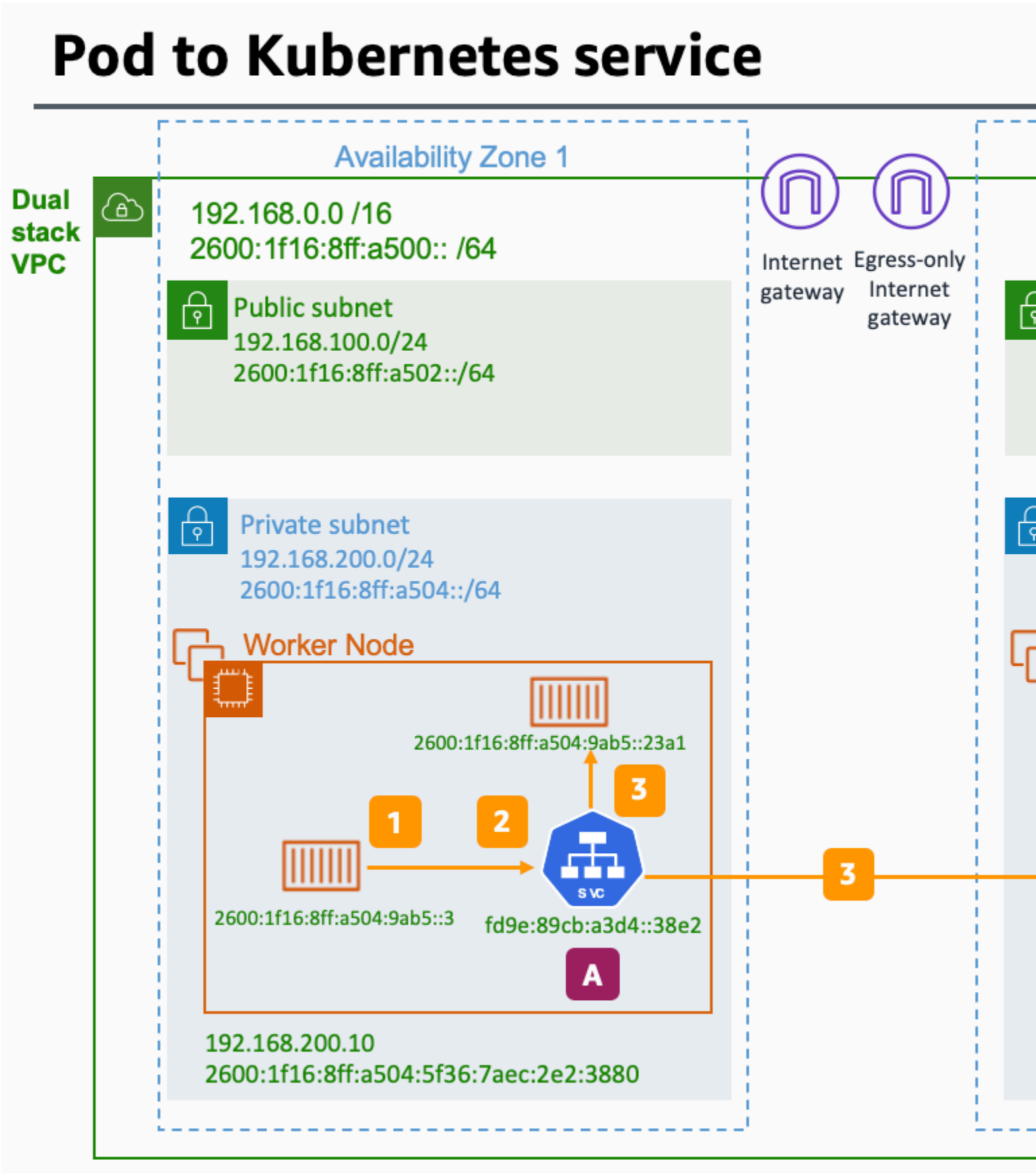
EKS/IPv6 Pod 还需要使用公有 IPv4 地址通过互联网连接到 IPv4 端点，才能实现类似的流程。下图描述了 IPv6 Pod 连接到集群边界外的 IPv4 端点的流程（可通过互联网路由）：


Pod to an IPv4 endpoint on the inte
IPv4
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
Route53
Elastic network interface (ENI)
2600:1f16:8ff:a504:9ab5::3
169.254.172.20
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880
1
2
3
4
5

在上图中，Pod 将对终端节点执行 DNS 查找，在收到 IPv4 “A” 响应后，Pod 的仅限节点的唯一 IPv4 地址通过源网络地址转换 (SNAT) 转换为连接到 EC2 的主网络接口的私有 IPv4 (VPC) 地址。 Worker-node然后，Pod IPv4 地址（源 IPv4：EC2 主 IP）被路由到 IPv4 NAT 网关，在那里将 EC2 主 IP 转换 (SNAT) 转换为有效的互联网可路由的 IPv4 公有 IP 地址（NAT 网关分配的公有 IP）。

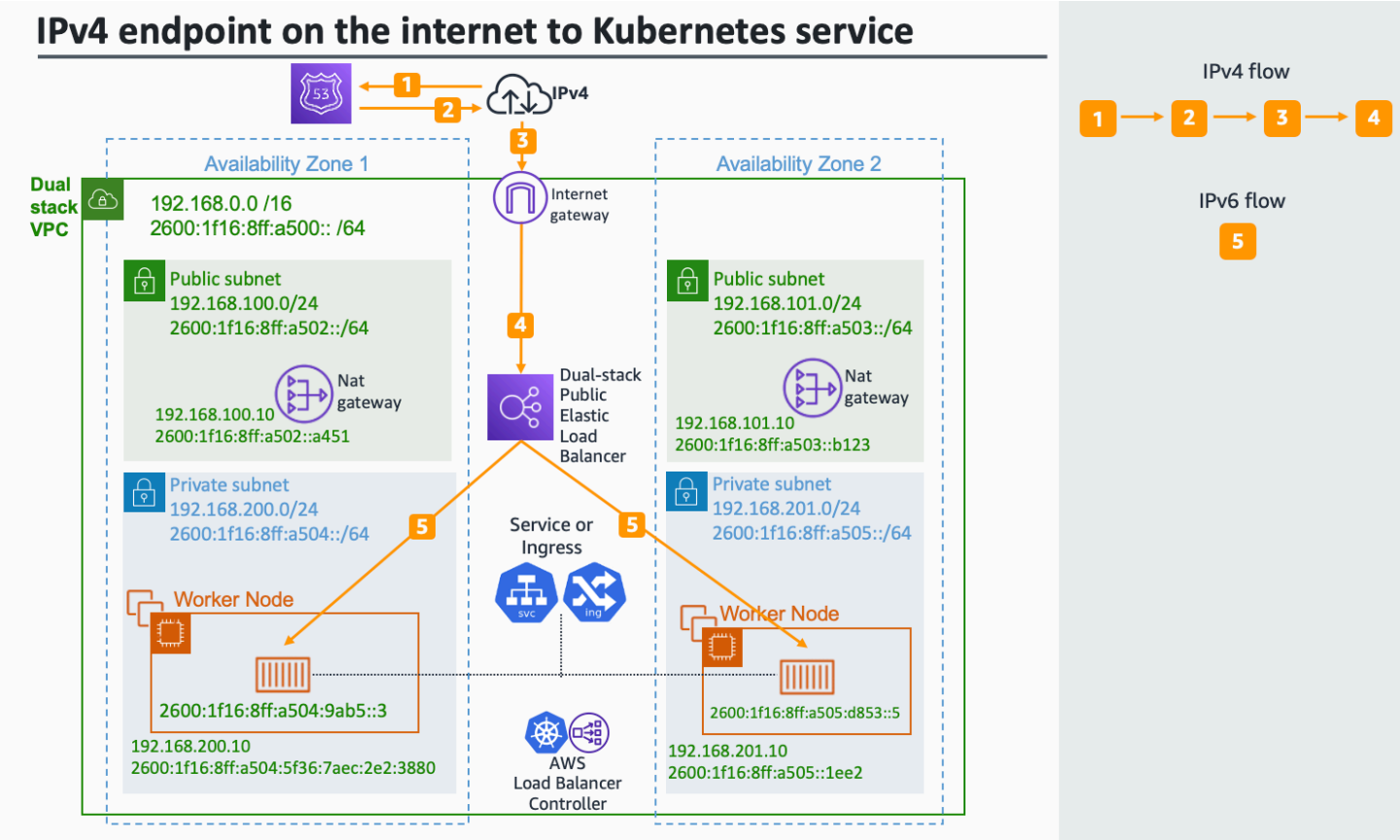
节点间的任何 Pod-to-Pod 通信都始终使用 IPv6 地址。VPC CNI 将 iptables 配置为处理 IPv6，同时阻止任何 IPv4 连接。

Kubernetes 服务将仅接收来自唯一本地 IPv6 单播地址 (ULA) 的 IPv6 地址 (clusterIP)。IPv6 集群的 ULA 服务 CIDR 是在 EKS 集群创建阶段自动分配的，无法修改。下图描述了 Pod 到 Kubernetes 的服务流程：


Pod to Kubernetes service
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Egress-only Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
2600:1f16:8ff:a504:9ab5::23a1
1
2
3
3
SVC
2600:1f16:8ff:a504:9ab5::3
fd9e:89cb:a3d4::38e2
A
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

使用 AWS 负载均衡器将服务公开到互联网。负载均衡器接收公有 IPv4 和 IPv6 地址，也就是双栈负载均衡器。对于访问 IPv6 集群 kubernetes 服务的 IPv4 客户端，负载均衡器会执行 IPv4 到 IPv6 的转换。

Amazon EKS 建议在私有子网中运行工作节点和 Pod。你可以在公有子网中创建公有负载均衡器，对私有子网中节点上运行的 Pod 的流量进行负载均衡。下图描绘了互联网 IPv4 用户访问基于 EKS/IPv6 入口的服务：

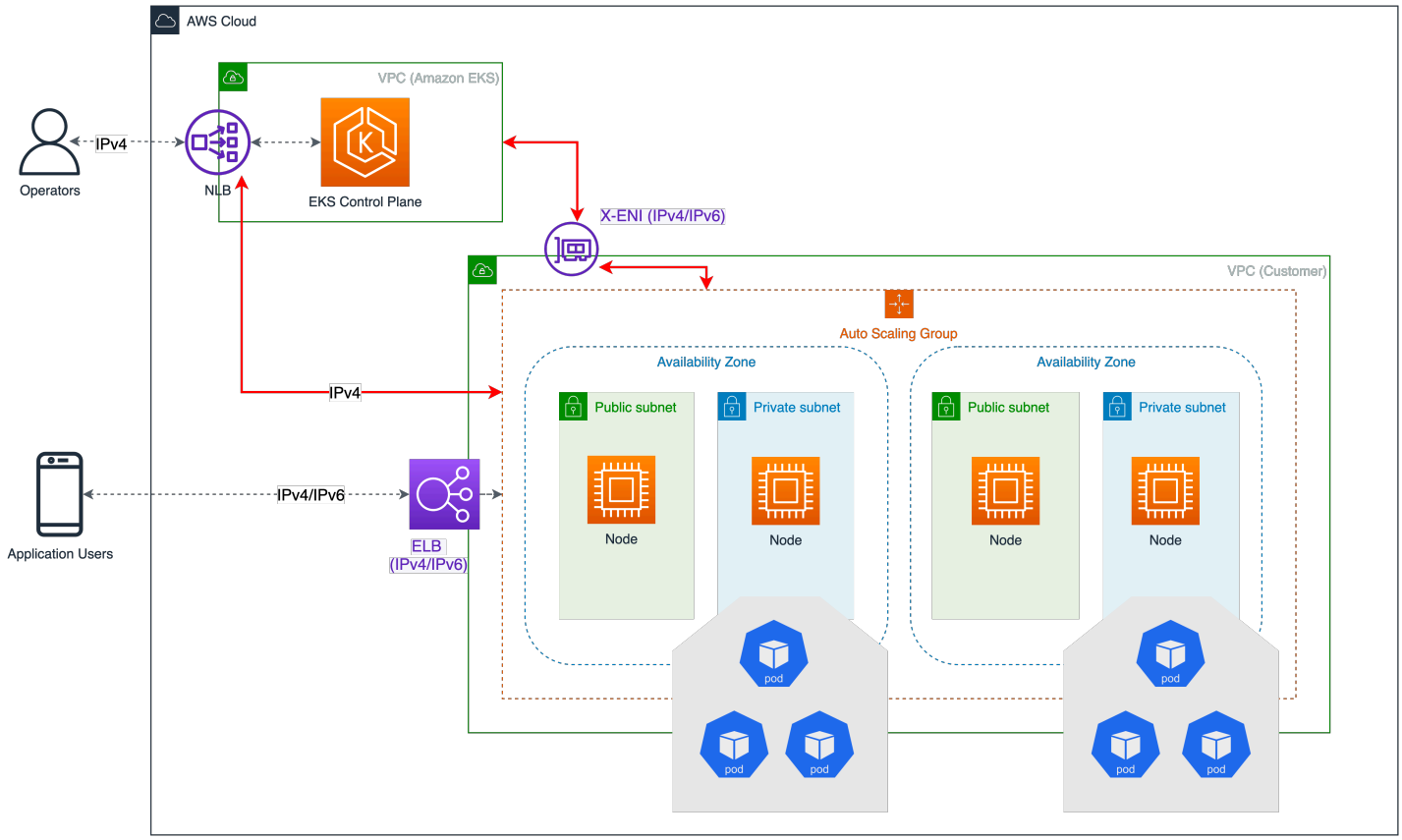



> **Note**
>
> 上述模式需要部署最新版本的 AWS 负载均衡器控制器

## EKS 控制平面数据平面通信

EKS 将在双堆栈模式下配置 Cross-Account ENI (IPv4/IPv6) ()。X-ENIsKubernetes 节点组件（例如 kubelet 和 kube-proxy）被配置为支持双堆栈。Kubelet 和 kube-proxy 在 HostNetwork 模式下运行，并绑定到连接到节点主网络接口的 IPv4 和 IPv6 地址。Kubernetes api 服务器通过基于 IPv6 的 Pod 和

节点组件进行通信。 X-ENIs Pod 通过与 api 服务器通信 X-ENIs，而 Pod 到 api 服务器的通信始终使用 IPv6 模式。




# 建议

## 基于计算资源的时间安排

一个 IPv6 前缀足以在单个节点上运行许多 Pod。这也有效地消除了对节点上最大 Pod 数量的 ENI 和 IP 限制。尽管 IPv6 消除了对 Max-Pods 的直接依赖，但在使用具有较小实例类型（例如 m5.large）的前缀附件时，您很可能会在耗尽实例的 IP 地址之前很久就耗尽实例的 CPU 和内存资源。如果您使用的是自管理节点组或带有自定义 AMI ID 的托管节点组，则必须手动设置 EKS 建议的最大 Pod 值。

您可以使用以下公式来确定在 IPv6 EKS 集群的节点上可以部署的最大 Pod 数量。

```
((Number of network interfaces for instance type (number of prefixes per network
 interface-1)* 16) + 2
```

```
((3 ENIs)_((10 secondary IPs per ENI-1)_ 16)) + 2 = 460 (real)
```

托管节点组会自动为您计算最大容器组（pod）数量。避免更改 EKS 对最大 Pod 数量的建议值，以避免由于资源限制而导致的 Pod 调度失败。

### 评估现有自定义网络的用途

如果当前启用了自定义联网，Amazon EKS 建议您重新评估您对 IPv6 的需求。如果您选择使用自定义网络来解决 IPv4 耗尽问题，则不再需要使用 IPv6。如果您使用自定义网络来满足安全要求，例如为节点和 Pod 设置单独的网络，我们建议您提交 EKS 路线图请求。

### 集群中的 Fargate 吊舱 EKS/IPv6

EKS 支持 IPv6 适用于在 Fargate 上运行的 Pod。在 Fargate 上运行的 Pod 将消耗 IPv6 和从 VPC CIDR 范围划分的 VPC 可路由私有 IPv4 地址（ ）。IPv4IPv6简而言之，你 EKS/Fargate 的 Pods 集群的宽密度将仅限于可用的 IPv4 和 IPv6 地址。建议调整双堆栈 subnets/VPC CIDR 的大小，以备将来的增长之需。如果底层子网不包含可用的 IPv4 地址，则无论 IPv6 可用地址如何，都无法安排新的 Fargate Pod。

### 部署 AWS Load Balancer 控制器 (LBC)

上游的树内 Kubernetes 服务控制器不支持 IPv6。我们建议使用最新版本的 AWS Load Balancer 控制器附加组件。LBC 只有在使用带有以下注释的相应的 kub service/ingress ernetes 定义后才会部署双栈 NLB 或双栈 ALB：和 `"alb.ingress.kubernetes.io/ip-address-type: dualstack"` `"alb.ingress.kubernetes.io/target-type: ip"`

## 自定义网络

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

默认情况下，Amazon VPC CNI 会为 Pod 分配从主子网中选择的 IP 地址。主子网是主 ENI 所连接的子网 CIDR，通常是的 node/host子网。

如果子网 CIDR 太小，CNI 可能无法获取足够的辅助 IP 地址来分配给你的 Pod。这是 EKS IPv4 集群面临的常见挑战。

自定义网络是解决这个问题的一种方法。

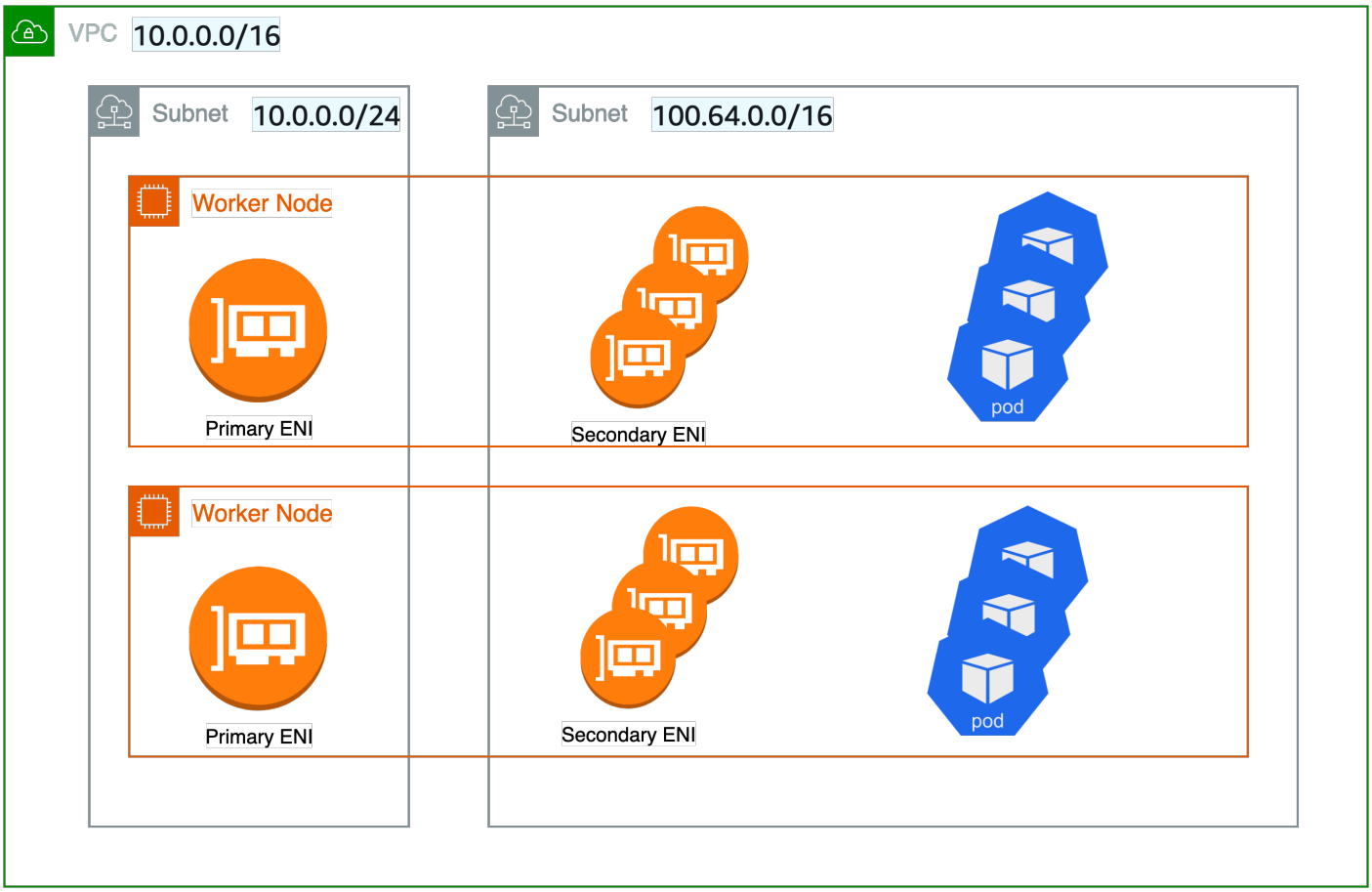
自定义网络通过从辅助 VPC 地址空间 (CIDR) 分配节点和 Pod IP 来解决 IP 耗尽问题。自定义网络支持支持 eniconFig 自定义资源。eniconFig 包括备用子网 CIDR 范围（由辅助 VPC CIDR 划分），以及 Pod 将属于的安全组。启用自定义联网后，VPC CNI 会在 eniconFig 下定义的子网中创建辅助弹性网卡。CNI 从 eniconFig CRD 中定义的 CIDR 范围为 Pod 分配一个 IP 地址。

由于自定义网络不使用主 ENI，因此您可以在节点上运行的最大 Pod 数量较低。主机网络 Pod 继续使用分配给主 ENI 的 IP 地址。此外，主 ENI 用于处理源网络转换并在节点之外路由 Pods 流量。

## 示例配置

虽然自定义网络将接受辅助 CIDR 范围的有效 VPC 范围，但我们建议您使用`100.64.0.0/10`共享地址空间 (RFC 6598) 中的 CIDR，因为与其他 RFC1918 范围相比，这些地址在企业环境中使用的可能性较小。例如，您可以`100.64.0.0/16`将其用作您的 VPC 的辅助 CIDR。有关您的 VPC 中允许和受限的 CIDR 区块关联的更多信息，请参阅 VPC 文档的 “VPC 和子网大小” 部分中的 IPv4 CIDR 区块关联限制。

如下图所示，工作节点的主弹性网络接口 (ENI) 仍使用主 VPC CIDR 范围（在本例中为 10.0.0）。0/16）但辅助 ENI 使用辅助 VPC CIDR 范围（在本例中为 100.64.0。 0/16)。现在，为了让 Pod 使用 100.64.0。 0/16 CIDR 范围，您必须将 CNI 插件配置为使用自定义网络。您可以按照此处记录的步骤进行操作。




如果您希望 CNI 使用自定义网络，请将AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG环境变量设置为。true

```
kubectl set env daemonset aws-node -n kube-system
 AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true
```

何时AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true，CNI 将从中定义的子网中分配 Pod IP 地址。ENIConfigENIConfig自定义资源用于定义调度 Pod 的子网。

```
apiVersion : crd.k8s.amazonaws.com/v1alpha1
kind : ENIConfig
metadata:
  name: us-west-2a
spec:
  securityGroups:
    - sg-0dff111a1d11c1c11
  subnet: subnet-011b111c1f11fdf11
```

创建ENIconfig自定义资源后，您将需要创建新的工作节点并耗尽现有节点。现有的工作节点和 Pod 将不受影响。

## 建议

### 在以下情况下使用自定义网络

如果您正在处理 IPv4 耗尽且还不能使用 IPv6，我们建议您考虑使用自定义网络。Amazon EKS 对 RFC6598 空间的支持使您能够将 Pod 扩展到 RFC1918 以外，以应对耗尽挑战。请考虑在自定义网络中使用前缀委派来增加节点上的 Pod 密度。

如果您需要在具有不同安全组要求的不同网络上运行 Pod，则可以考虑自定义联网。启用自定义网络后，Pod 使用的子网或安全组与 eniconFig 中的定义不同，而不是节点的主网络接口。

自定义网络确实是部署多个 EKS 集群和应用程序以连接本地数据中心服务的理想选择。您可以增加 VPC 中 EKS 可访问的私有地址数量 (RFC1918)，以获取 Amazon Elastic Load Balancing 等服务 NAT-GW，同时为跨多个集群的 Pod 使用不可路由的 CG-NAT 空间。使用传输网关和共享服务 VPC（包括跨多个可用区的 NAT 网关以实现高可用性）的自定义联网使您能够提供可扩展且可预测的流量。这篇博客文章描述了一种架构模式，这是使用自定义网络将 EKS Pod 连接到数据中心网络的最推荐方法之一。

### 在以下情况下避免使用自定义网络

已准备好实施 IPv6

自定义网络可以缓解 IP 耗尽问题，但需要额外的运营开销。如果您当前正在部署双堆栈 (IPv4/IPv6) VPC，或者您的计划包括 IPv6 支持，我们建议改为实施 IPv6 集群。您可以设置 IPv6 EKS 集群并迁移您的应用程序。在 IPv6 EKS 集群中，Kubernetes 和 Pod 都获得 IPv6 地址，并且可以与 IPv4 和 IPv6 端点进行进出通信。请查看运行 IPv6 EKS 集群的最佳实践。

已耗尽的 CG-NAT 空间

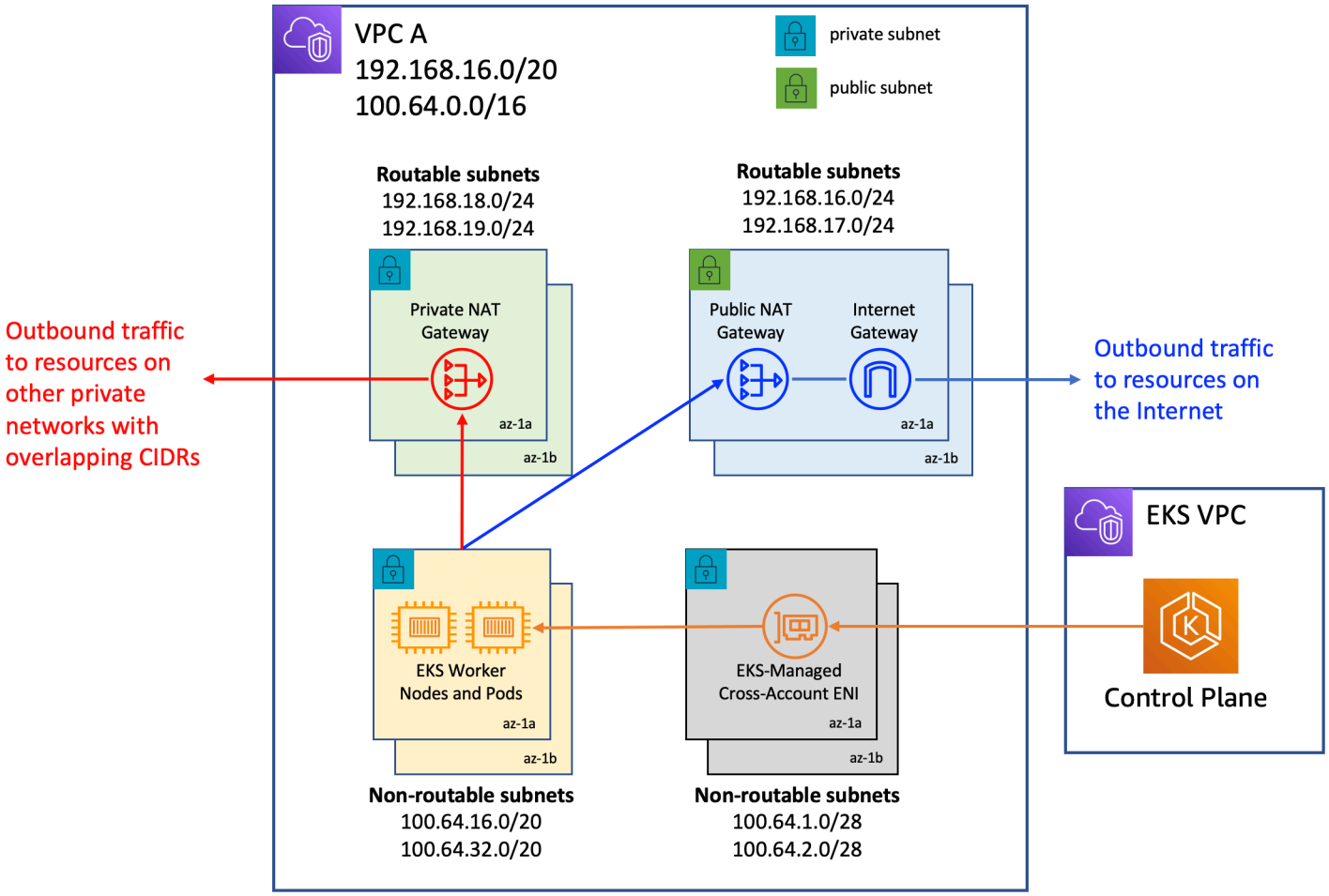
此外，如果您目前正在使用该 CG-NAT 空间中的 CIDR，或者无法将辅助 CIDR 与集群 VPC 关联起来，则可能需要探索其他选项，例如使用备用 CNI。我们强烈建议您获得商业支持或拥有内部知识来调试和向开源 CNI 插件项目提交补丁。有关更多详细信息，请参阅备用 CNI 插件用户指南。

使用私有 NAT 网关

Amazon VPC 现在提供私有 NAT 网关功能。Amazon 的私有 NAT 网关使私有子网中的实例能够使用重叠 CIDR 连接到其他 VPC 和本地网络。考虑使用本博客文章中描述的方法使用私有 NAT 网关来克

服 CIDR 重叠导致的 EKS 工作负载的通信问题，这是我们的客户表达的重大抱怨。自定义网络无法单独解决重叠的 CIDR 问题，这增加了配置难题。

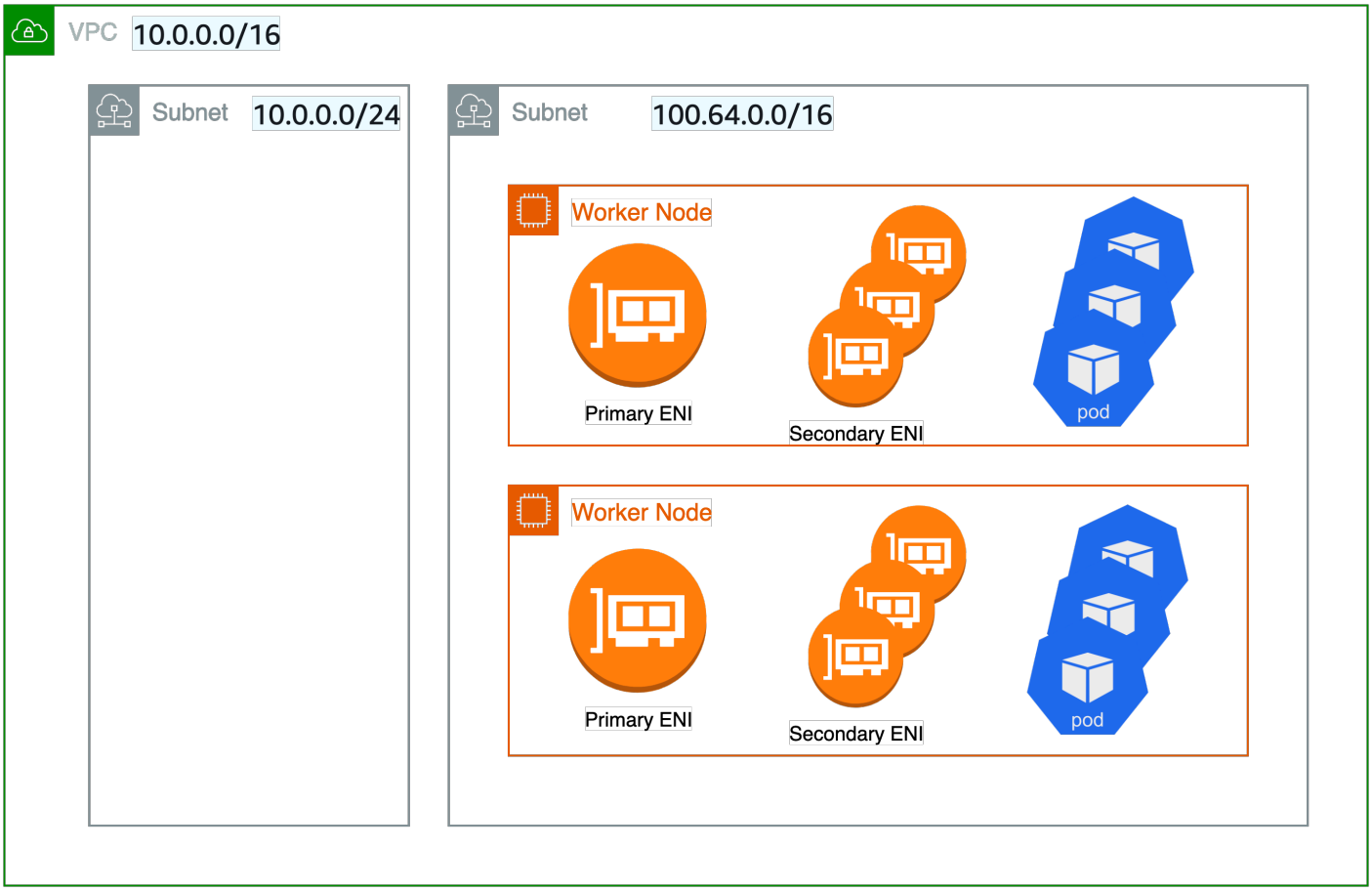
本博客文章实施中使用的网络架构遵循了 Amazon VPC 文档中启用重叠网络间通信下的建议。如这篇博客文章所示，您可以将私有 NAT 网关与 RFC6598 地址结合使用的范围扩展到管理客户的私有 IP 耗尽问题。EKS 集群、工作节点部署在不可路由的 100.64.0 中。 0/16 VPC 辅助 CIDR 范围，而私有 NAT 网关、NAT 网关部署到可路由的 RFC1918 CIDR 范围。该博客介绍了如何使用传输网关连接 VPC，以促进不可路由 CIDR 范围重叠的 VPC 之间的通信。对于 VPC 不可路由地址范围内的 EKS 资源需要与地址范围不重叠的其他 VPC 通信的用例，客户可以选择使用 VPC 对等互连来互连这些 VPC。这种方法可以节省成本，因为通过 VPC 对等连接在可用区内传输的所有数据现在都是免费的。




节点和 Pod 的独特网络

如果您出于安全原因需要将节点和 Pod 隔离到特定网络，我们建议您将节点和 Pod 从更大的辅助 CIDR 块（例如 100.64.0）部署到子网中。 0/8)。在您的 VPC 中安装新 CIDR 后，您可以使用辅助 CIDR 部署另一个节点组并耗尽原始节点，从而自动将 Pod 重新部署到新的工作节点上。有关如何实现这一点的更多信息，请参阅此博客文章。

下图所示的设置中不使用自定义网络。相反，Kubernetes 工作节点部署在您的 VPC 的辅助 VPC CIDR 范围（例如 100.64.0）的子网上。 0/10。您可以保持 EKS 集群运行（控制平面将保留在原始集群上 subnet/s），但节点和 Pod 将移至辅助节点 subnet/s。这是另一种缓解 VPC 中 IP 耗尽危险的非常规技术。我们建议在将 pod 重新部署到新的工作节点之前，先耗尽旧节点。




## 使用可用区标签自动配置

你可以让 Kubernetes 自动为工作节点可用区 (AZ) 应用相应的 eniconFig。

Kubernetes 会自动将标签[topology.kubernetes.io/zone](topology.kubernetes.io/zone)添加到你的工作节点中。当每个可用区只有一个辅助子网（备用 CIDR）时，Amazon EKS 建议使用可用区作为 ENI 配置名称。然后，您可以将用于发现 ENI 配置名称的标签设置为`topology.kubernetes.io/zone`。请注意，标签已`failure-domain.beta.kubernetes.io/zone`被弃用并替换为标签`topology.kubernetes.io/zone`。

1. 将`name`字段设置为您的 VPC 的可用区。
2. 通过以下命令启用自动配置

3. 通过以下命令设置配置标签

```
kubectl set env daemonset aws-node -n kube-system
 "AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true"
kubectl set env daemonset aws-node -n kube-system
 "ENI_CONFIG_LABEL_DEF=topology.kubernetes.io/zone"
```

如果每个可用区有多个辅助子网，则需要创建一个特定的ENI_CONFIG_LABEL_DEF辅助子网。您可以考虑使用自定义 eniconFig 名称（ENI_CONFIG_LABEL_DEF例如k8s.amazonaws.com/eniConfig和）配置为和标记节点。k8s.amazonaws.com/eniConfig=us-west-2a-subnet-1k8s.amazonaws.com/eniConfig=us-west-2a-subnet-2

## 配置辅助网络时替换 Pod

启用自定义联网不会修改现有节点。自定义联网是一种破坏性行为。我们建议不要在启用自定义联网后滚动替换集群中的所有工作节点，而是使用调用 Lambda 函数的自定义资源更新 E KS 入门指南中的 AWS CloudFormation 模板，使用环境变量更新 aws-node Daemonset，以便在配置工作节点之前启用自定义联网。

如果在切换到自定义 CNI 网络功能之前，你的集群中有任何正在运行 Pods 的节点，你应该封锁并排空节点，以优雅地关闭 Pod，然后终止节点。只有与 eniconFig 标签或注释匹配的新节点才使用自定义网络，因此可以从辅助 CIDR 为调度在这些新节点上的 Pod 分配一个 IP。

## 计算每个节点的最大 Pod 数

由于不再使用节点的主 ENI 来分配 Pod IP 地址，因此在给定 EC2 实例类型上可以运行的 Pod 数量有所减少。要解决此限制，可以在自定义网络中使用前缀分配。通过前缀分配，辅助 ENI 上的每个辅助 IP 都被替换为 /28 前缀。

考虑使用自定义网络的 m5.large 实例的最大 Pod 数量。

在不分配前缀的情况下可以运行的最大 Pod 数为 29

- `3 ENIs - 1) * (10 secondary IPs per ENI - 1 + 2 = 20`

启用前缀附件会将 Pod 的数量增加到 290 个。

- `(3 ENIs - 1) * ((10 secondary IPs per ENI - 1) * 16 + 2 = 290`

但是，我们建议将最大容量设置为 110 而不是 290，因为该实例的虚拟 CPU 数量相当少。对于较大的实例，EKS 建议 pod 的最大值为 250。使用具有较小实例类型（例如 m5.large）的前缀附件时，您可能会在实例的 IP 地址之前耗尽实例的 CPU 和内存资源。

> **Note**
>
> 当 CNI 前缀为 ENI 分配一个 /28 前缀时，它必须是一个连续的 IP 地址块。如果生成前缀的子网高度分散，则前缀连接可能会失败。您可以通过为集群创建新的专用 VPC 或为子网预留一组 CIDR 专门用于前缀附件来缓解这种情况。有关此主题的更多信息，请访问子网 CIDR 预留。

### 确定 CG-NAT 空间的现有使用情况

自定义网络允许您缓解 IP 耗尽问题，但它无法解决所有挑战。如果您已经在为集群使用 CG-NAT 空间，或者根本无法将辅助 CIDR 与集群 VPC 相关联，我们建议您探索其他选项，例如使用备用 CNI 或迁移到 IPv6 集群。

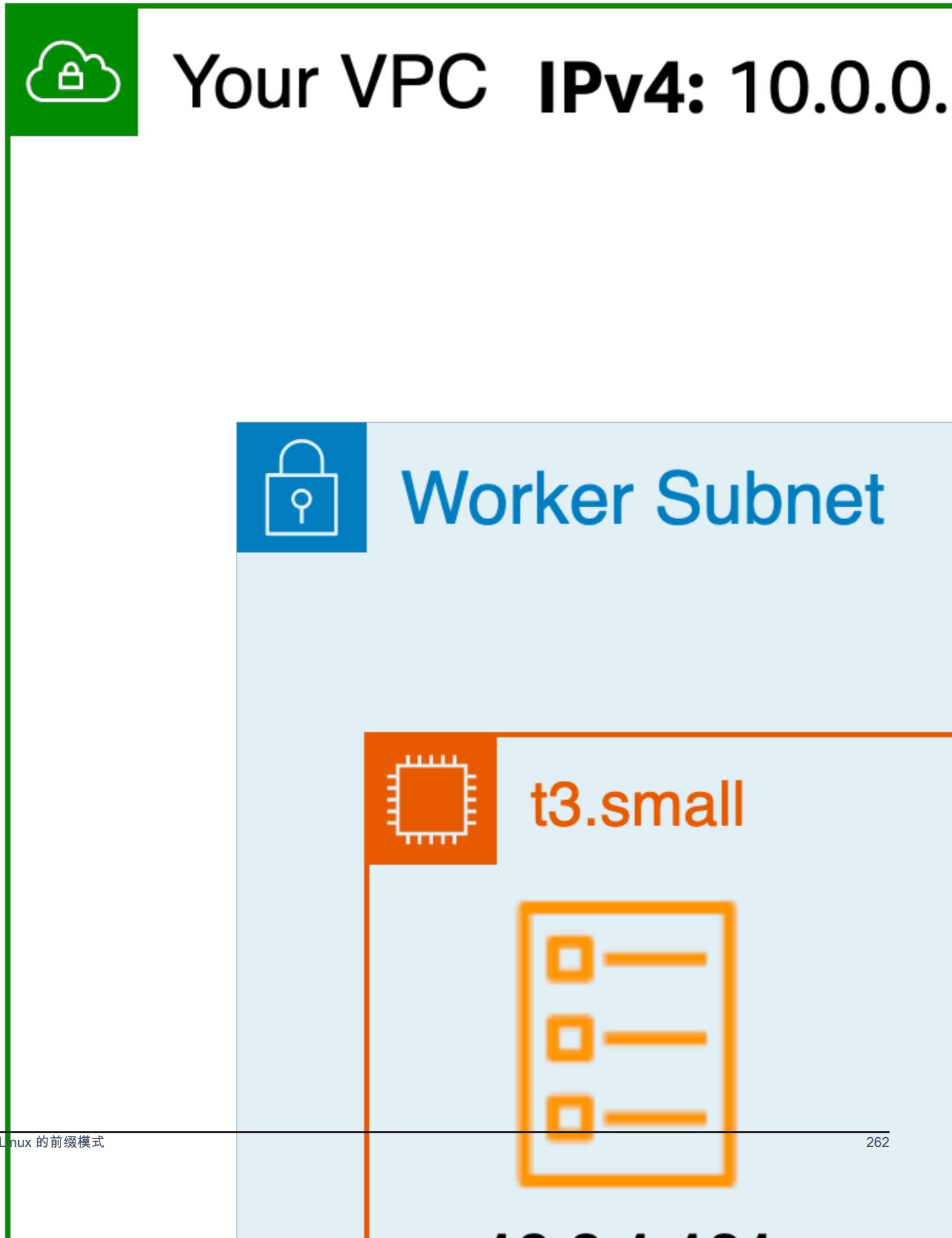
## 适用于 Linux 的前缀模式

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Amazon VPC CNI 为 Amazon EC2 网络接口分配网络前缀，以增加节点可用的 IP 地址数量并增加每个节点的容器密度。您可以将版本 1.9.0 或更高版本的 Amazon VPC CNI 插件配置为分配 IPv4 和 IPv6 CIDR，而不是为网络接口分配单独的辅助 IP 地址。

IPv6 群集上默认启用前缀模式，并且是唯一支持的选项。VPC CNI 为弹性网卡上的插槽分配一个 /80 IPv6 前缀。有关更多信息，请参阅本指南的 IPv6 部分。

在前缀分配模式下，每种实例类型的最大弹性网络接口数量保持不变，但您现在可以将 Amazon VPC CNI 配置为分配 /28（16 个 IP 地址）IPv4 地址前缀，而不是为网络接口上的插槽分配单独的 IPv4 地址。当设置`ENABLE_PREFIX_DELEGATION`为 true 时，VPC CNI 会从分配给 ENI 的前缀中为 Pod 分配一个 IP 地址。请按照 EKS 用户指南中提及的说明启用前缀 IP 模式。

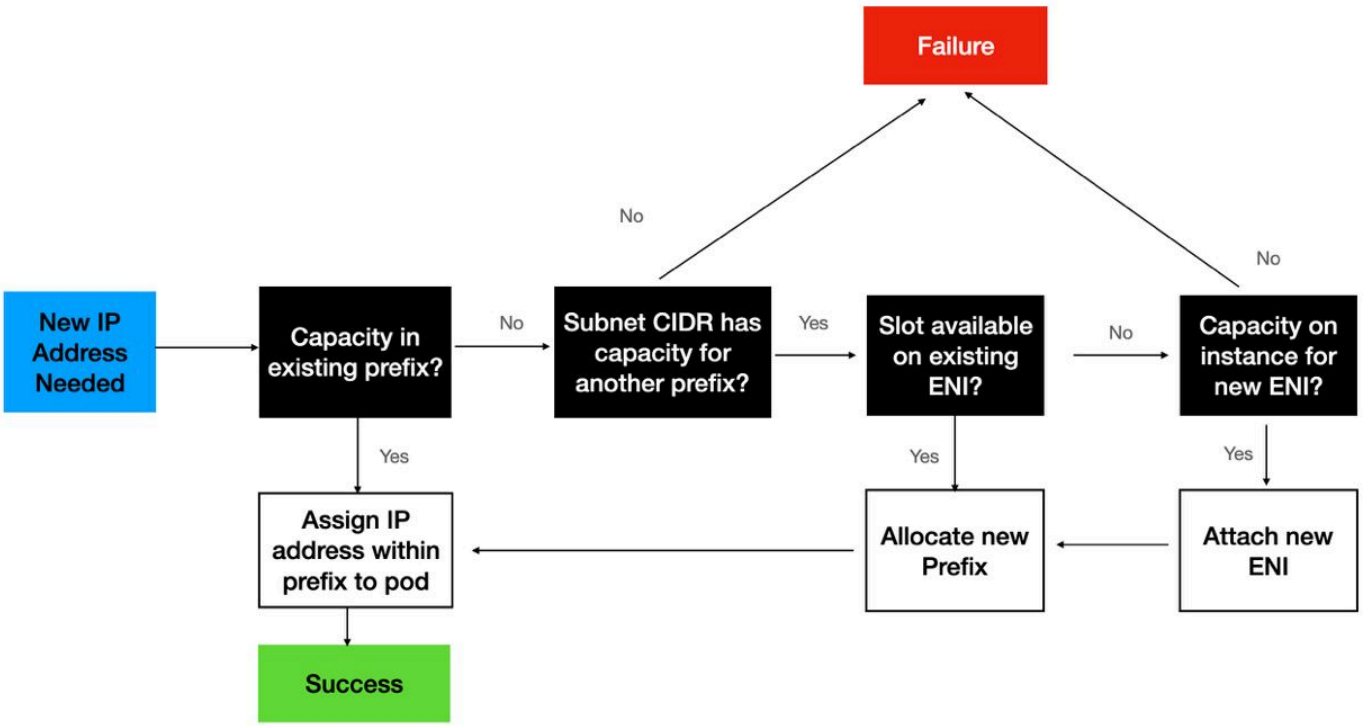

Your VPC IPv4: 10.0.0.
Worker Subnet
t3.small

您可以分配给网络接口的 IP 地址的最大数量取决于实例类型。您分配给网络接口的每个前缀都算作一个 IP 地址。例如，`c5.large` 实例具有每个网络接口 `10` 个 IPv4 地址的限制。此实例的每个网络接口都有一个主 IPv4 地址。如果网络接口没有辅助 IPv4 地址，则您最多可以为网络接口分配 9 个前缀。您每为网络接口多分配一个 IPv4 地址，则您就可以为网络接口少分配一个前缀。查看 AWS EC2 文档，了解每种实例类型的每个网络接口的 IP 地址以及为网络接口分配前缀。

在工作节点初始化期间，VPC CNI 会为主 ENI 分配一个或多个前缀。CNI 预先分配了一个前缀，通过维护温池，可以更快地启动 pod。可以通过设置环境变量来控制要在温池中保存的前缀数量。

- `WARM_PREFIX_TARGET`，即要分配的超出当前需求的前缀数量。
- `WARM_IP_TARGET`，要分配的超出当前需求的 IP 地址数量。
- `MINIMUM_IP_TARGET`，任何时候可用的 IP 地址的最小数量。
- `WARM_IP_TARGETMINIMUM_IP_TARGET`如果设置则会覆盖`WARM_PREFIX_TARGET`。

随着计划的 Pod 越多，将为现有 ENI 请求额外的前缀。首先，VPC CNI 尝试为现有 ENI 分配新的前缀。如果 ENI 已满负荷，VPC CNI 会尝试向该节点分配新的弹性网卡。在达到最大 ENI 上限（由实例类型定义）之前，将附加新的 ENI。附加新的 ENI 时，ipamd 将分配维护`WARM_PREFIX_TARGETWARM_IP_TARGET`、和`MINIMUM_IP_TARGET`设置所需的一个或多个前缀。



# 建议

## 在以下情况下使用前缀模式

如果您在工作节点上遇到 Pod 密度问题，请使用前缀模式。为避免 VPC CNI 错误，我们建议在迁移到前缀模式之前，检查子网中是否有 /28 前缀的连续地址块。有关子网预留的详细信息，请参阅 “使用子网预留避免子网碎片 (IPv4)” 部分。

为了向后兼容，最大容量限制设置为支持辅助 IP 模式。要增加 pod 密度，请将 Kubelet 的`max-pods--use-max-pods=false`值指定为节点的用户数据。有关更多信息，请参阅 Amazon EKS 用户指南中的如何确定 MaxPods。有关用户数据的示例，请参阅 EKS 用户指南。

```
./max-pods-calculator.sh --instance-type m5.large --cni-version ``1.9``.0 --cni-prefix-delegation-enabled
```

前缀分配模式对于 CNI 自定义网络的用户尤其重要，在这种网络中，主网卡不用于 pod。通过前缀分配，您仍然可以在几乎每种 Nitro 实例类型上附加更多 IP，即使没有将主 ENI 用于 pod 也是如此。

## 在以下情况下避免使用前缀模式

如果您的子网非常分散，且可用的 IP 地址不足，无法创建 /28 前缀，请避免使用前缀模式。如果生成前缀的子网碎片化（一个经常使用的子网，次要 IP 地址分散），则前缀连接可能会失败。可以通过创建新子网并保留前缀来避免此问题。

在前缀模式下，分配给工作节点的安全组由 Pod 共享。如果您有通过在共享计算资源上运行具有不同网络安全要求的应用程序来实现合规性的安全要求，可以考虑为 Pod 使用安全组。

## 在同一个节点组中使用相似的实例类型

您的节点组可能包含多种类型的实例。如果实例的最大 Pod 数量较低，则该值将应用于节点组中的所有节点。考虑在节点组中使用相似的实例类型，以最大限度地提高节点使用率。我们建议配置 node.kubernetes。 io/instance-如果您使用 Karp enter 进行自动节点扩展，请在配置器 API 的要求部分中键入。

> **Warning**
>
> 特定节点组中所有节点的最大容量由该节点组中任何单个实例类型的最小最大容器数量定义。

### 配置 WARM_PREFIX_TARGET #保存 IPv4 地址

安装清单的默认值WARM_PREFIX_TARGET为 1。在大多数情况下，建议的值 1 可以很好地兼顾快速的 pod 启动时间，同时最大限度地减少分配给实例的未使用的 IP 地址。WARM_PREFIX_TARGET

如果您需要进一步节省每个节点的 IPv4 地址，请使用WARM_IP_TARGET和MINIMUM_IP_TARGET设置（配置WARM_PREFIX_TARGET后会覆盖这些地址）。通过将值设置WARM_IP_TARGET为小于 16，可以防止 CNI 保留全部多余的前缀。

### 与附加新 ENI 相比，更喜欢分配新的前缀

与创建新的 ENI 并将其附加到实例相比，为现有 ENI 分配额外前缀更快的 EC2 API 操作。使用前缀可以提高性能，同时节俭 IPv4 地址分配。附加前缀通常在不到一秒钟的时间内完成，而连接新的 ENI 可能需要长达 10 秒的时间。对于大多数用例，CNI 在前缀模式下运行时，每个工作节点只需要一个 ENI。如果您负担得起（在最坏的情况下）每个节点多达 15 个未使用的 IP，我们强烈建议您使用更新的前缀分配网络模式，并实现随之而来的性能和效率提升。

### 使用子网预留避免子网碎片 (IPv4)

当 EC2 为 ENI 分配 /28 IPv4 前缀时，它必须是您子网中连续的 IP 地址块。如果生成前缀的子网是碎片化的（一个高度使用的子网，具有分散的辅助 IP 地址），则前缀连接可能会失败，并且您将在 VPC CNI 日志中看到以下错误消息：

```
failed to allocate a private IP/Prefix address: InsufficientCidrBlocks: There are not
 enough free cidr blocks in the specified subnet to satisfy the request.
```

为了避免分段并有足够的连续空间来创建前缀，您可以使用 VP C 子网 CIDR 预留在子网内保留 IP 空间供前缀专用。创建预留后，VPC CNI 插件将调用 EC2 API 来分配从预留空间自动分配的前缀。

建议创建一个新子网，为前缀预留空间，并使用 VPC CNI 为在该子网中运行的工作节点启用前缀分配。如果新子网仅专用于在启用了 VPC CNI 前缀分配的 EKS 集群中运行的 Pod，则可以跳过前缀预留步骤。

### 避免降级 VPC CNI

前缀模式适用于 VPC CNI 版本 1.9.0 及更高版本。启用前缀模式并为 ENI 分配前缀后，必须避免将 Amazon VPC CNI 附加组件降级到低于 1.9.0 的版本。如果您决定降级 VPC CNI，则必须删除并重新创建节点。

### 在过渡到前缀委派期间替换所有节点

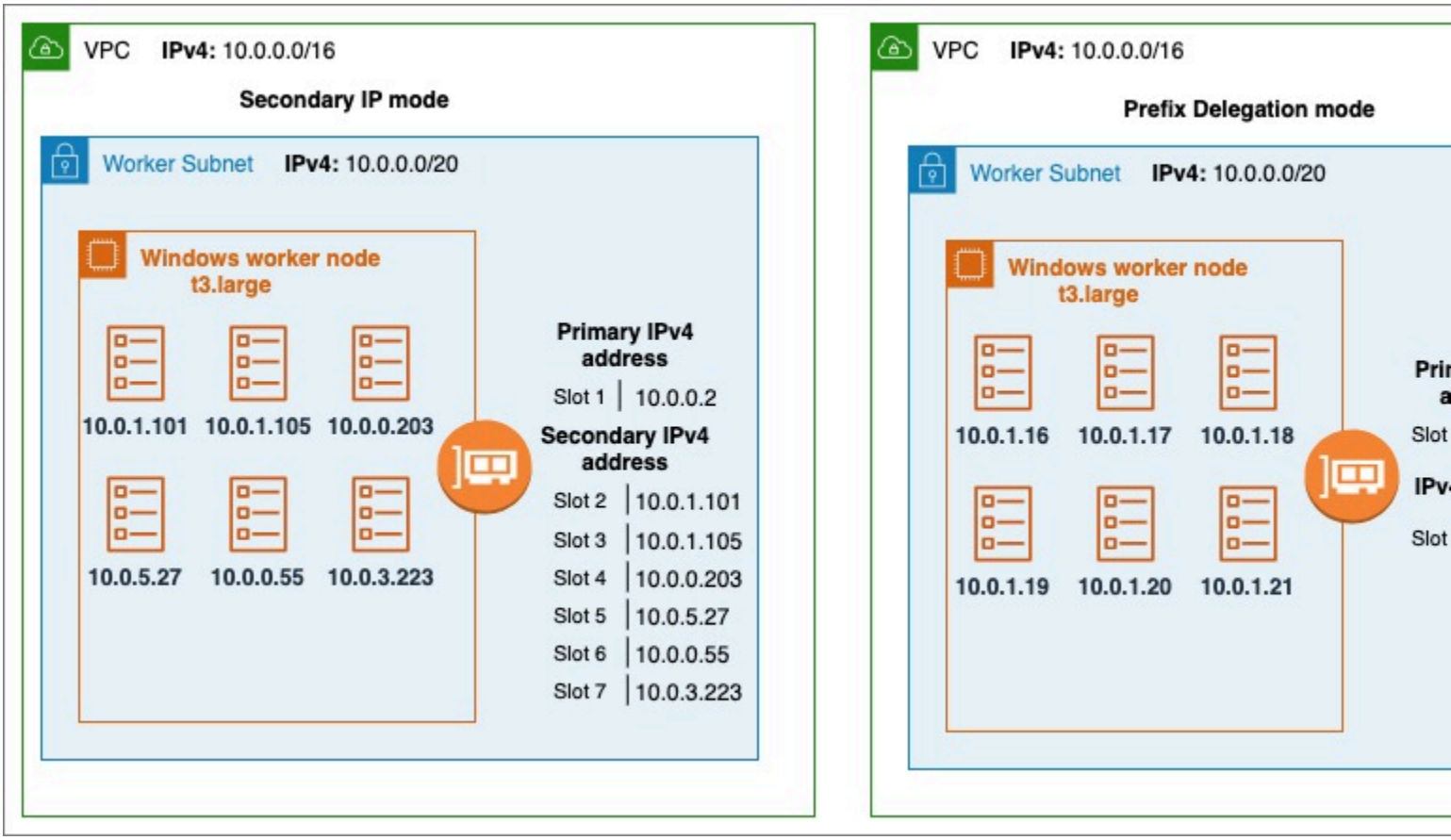
强烈建议您创建新的节点组以增加可用 IP 地址的数量，而不是滚动替换现有的工作节点。封锁并耗尽所有现有节点，以安全地驱逐所有现有 Pod。为了防止服务中断，我们建议在生产集群上为关键工作负载实施 Pod 中断预算。新节点上的 Pod 将从分配给 ENI 的前缀中分配一个 IP。确认 Pod 正在运行后，您可以删除旧的节点和节点组。如果您使用的是托管节点组，请按照此处提到的步骤安全地删除节点组。

## 适用于 Windows 的前缀模式

在 Amazon EKS 中，默认情况下，VP C 资源控制器会为在 Windows 主机上运行的每个 Pod 分配一个辅助 IP 地址。此 IP VPC-routable 地址是从主机的子网分配的地址。在 Linux 上，连接到实例的每个 ENI 都有多个插槽，可以用辅助 IP 地址或 /28 CIDR（前缀）填充这些插槽。但是，Windows 主机仅支持单个 ENI 及其可用插槽。即使有大量 IP 地址可供分配，也只能使用辅助 IP 地址可以人为地限制你可以在 Windows 主机上运行的 pod 数量。

为了提高 Windows 主机上的容器密度，尤其是在使用较小的实例类型时，您可以为 Windows 节点启用前缀委派。启用前缀委派后，会将 /28 IPv4 前缀分配给 ENI 插槽，而不是辅助 IP 地址。可以通过将`enable-windows-prefix-delegation: "true"`条目添加到`amazon-vpc-cni`配置映射来启用前缀委托。这与配置映射相同，你需要在其中设置`enable-windows-ipam: "true"`条目才能启用 Windows 支持。

请按照 EKS 用户指南中提及的说明为 Windows 节点启用前缀委派模式。



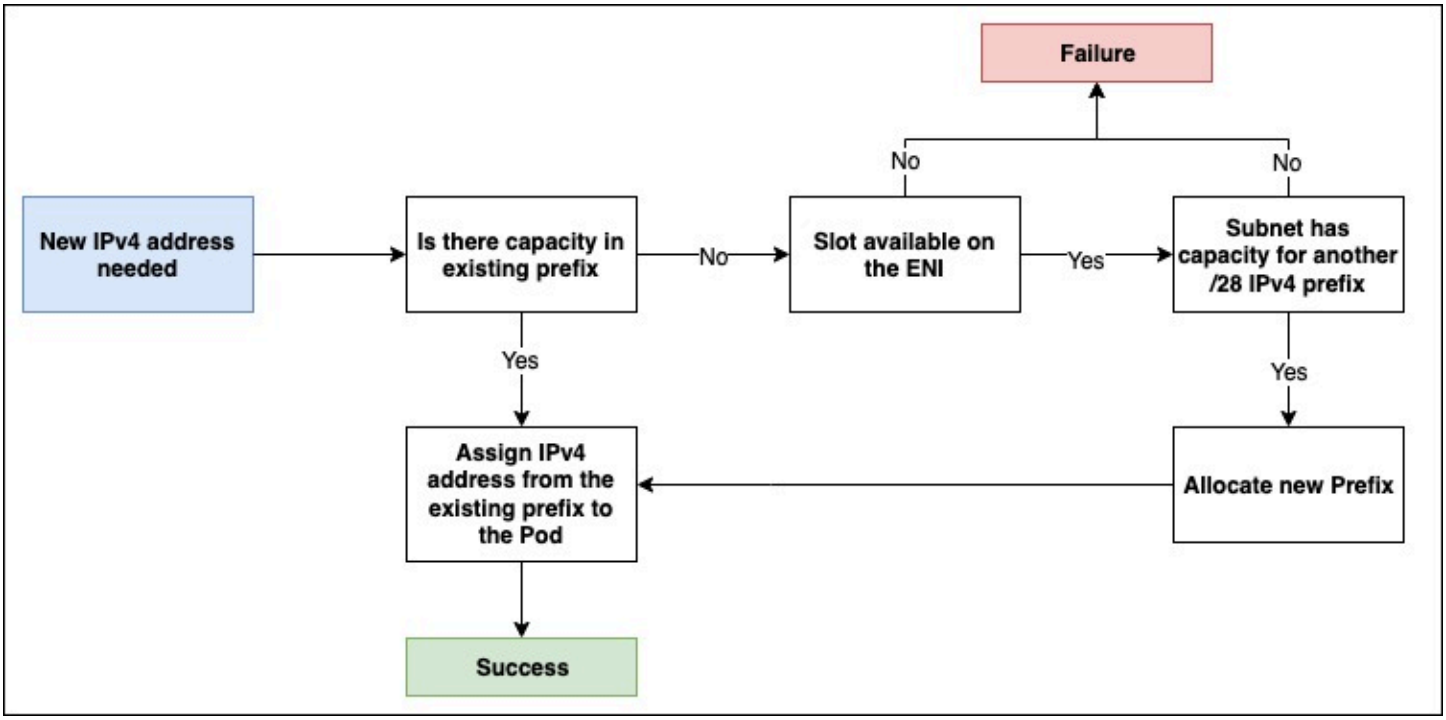

图：辅助 IP 模式与前缀委派模式的比较

您可以为网络接口分配的最大 IP 地址数取决于实例类型及其大小。分配给网络接口的每个前缀都会占用一个可用插槽。例如，`c5.large`实例对每个网络接口的10插槽数有限制。网络接口上的第一个插槽总是被接口的主要 IP 地址占用，剩下 9 个插槽用于前缀 and/or 辅助 IP 地址。如果为这些插槽分配了前缀，则节点可以支持 (9 * 16) 144 个 IP 地址，而如果为它们分配了辅助 IP 地址，则只能支持 9 个 IP 地址。有关更多信息，请参阅有关每种实例类型的每个网络接口的 IP 地址以及为网络接口分配前缀的文档。

在工作节点初始化期间，VPC 资源控制器会为主 ENI 分配一个或多个前缀，通过维护 IP 地址的热池，从而更快地启动 Pod。通过在配置映射中设置以下配置参数，可以控制要在温池中`amazon-vpc-cni`保存的前缀数量。

- `warm-prefix-target`，即要分配的超出当前需求的前缀数量。
- `warm-ip-target`，要分配的超出当前需求的 IP 地址数量。
- `minimum-ip-target`，任何时候可用的 IP 地址的最小数量。
- `warm-ip-target` and/or `minimum-ip-target`如果设置则会覆盖`warm-prefix-target`。

随着节点上调度更多的 Pod，将为现有 ENI 请求额外的前缀。在节点上调度 Pod 时，VPC 资源控制器将首先尝试从节点上的现有前缀中分配一个 IPv4 地址。如果这不可能，则只要子网具有所需的容量，就会请求新的 IPv4 前缀。




图：为 Pod 分配 IPv4 地址期间的工作流程

## 建议

### 在以下情况下使用前缀委托

如果您在工作节点上遇到 Pod 密度问题，请使用前缀委派。为避免错误，我们建议在迁移到前缀模式之前，检查子网中是否有 /28 前缀的连续地址块。有关子网预留的详细信息，请参阅 “使用子网预留避免子网碎片 (IPv4)” 部分。

默认情况下，Windows 节点`max-pods`上的设置为`110`。对于绝大多数实例类型来说，这应该足够了。如果您想增加或减少此限制，请在用户数据中的 bootstrap 命令中添加以下内容：

```
-KubeletExtraArgs '--max-pods=example-value'
```

有关 Windows 节点的引导程序配置参数的更多详细信息，请访问此处的文档。

### 在以下情况下避免前缀委托

如果您的子网非常分散，且可用的 IP 地址不足，无法创建 /28 前缀，请避免使用前缀模式。如果生成前缀的子网碎片化（一个经常使用的子网，次要 IP 地址分散），则前缀连接可能会失败。可以通过创建新子网并保留前缀来避免此问题。

### 配置前缀委派参数以保存 IPv4 地址

`warm-prefix-target`、`warm-ip-target`，并`minimum-ip-target`可用于微调预缩放和带前缀的动态缩放行为。默认情况下，使用以下值：

```
warm-ip-target: "1"
minimum-ip-target: "3"
```

通过微调这些配置参数，您可以在节省 IP 地址和确保减少因分配 IP 地址而导致的 Pod 延迟之间取得最佳平衡。有关这些配置参数的更多信息，请访问此处的文档。

### 使用子网预留避免子网碎片 (IPv4)

当 EC2 为 ENI 分配 /28 IPv4 前缀时，它必须是您子网中连续的 IP 地址块。如果生成前缀的子网是碎片化的（具有分散的辅助 IP 地址的高度使用子网），则前缀连接可能会失败，您将看到以下节点事件：

```
InsufficientCidrBlocks: The specified subnet does not have enough free cidr blocks to
 satisfy the request
```

为避免分段并有足够的连续空间来创建前缀，请使用 VP C 子网 CIDR 预留在子网内保留 IP 空间供前缀专用。创建预留后，保留区块中的 IP 地址将不会分配给其他资源。这样，VPC 资源控制器将能够在对节点 ENI 的分配调用期间获得可用的前缀。

建议创建一个新子网，为前缀预留空间，并为在该子网中运行的工作节点启用前缀分配。如果新子网仅专用于在启用前缀委派的 EKS 集群中运行的 Pod，则可以跳过前缀预留步骤。

### 从辅助 IP 模式迁移到前缀委派模式时替换所有节点，反之亦然

强烈建议您创建新的节点组以增加可用 IP 地址的数量，而不是滚动替换现有的工作节点。

使用自我管理的节点组时，过渡步骤为：

- 增加集群中的容量，以便新节点能够容纳您的工作负载

- Enable/Disable 适用于 Windows 的前缀委派功能
- 封锁并耗尽所有现有节点，以安全地驱逐所有现有 Pod。为了防止服务中断，我们建议在生产集群上为关键工作负载实施 Pod 中断预算。
- 确认 Pod 正在运行后，您可以删除旧的节点和节点组。新节点上的 Pod 将从分配给节点 ENI 的前缀中分配一个 IPv4 地址。

使用托管节点组时，过渡步骤为：

- Enable/Disable 适用于 Windows 的前缀委派功能
- 使用此处提到的步骤更新节点组。其执行的步骤与上述类似，但由 EKS 管理。

> **Warning**
>
> 以相同模式运行节点上的所有 Pod

对于 Windows，我们建议你避免同时在辅助 IP 模式和前缀委派模式下运行 Pod。当你在运行 Windows 工作负载时从辅助 IP 模式迁移到前缀委派模式时，可能会出现这种情况，反之亦然。

虽然这不会影响你正在运行的 Pod，但节点的 IP 地址容量可能存在不一致。例如，假设一个 t3.xlarge 节点有 14 个用于存放辅助 IPv4 地址的插槽。如果你运行的是 10 个 Pod，那么 ENI 上的 10 个插槽将被辅助 IP 地址占用。启用前缀委派后，向 kube-api 服务器通告的容量为（14 个插槽 * 每个前缀 16 个 IP 地址）244，但当时的实际容量为（剩下 4 个插槽 * 每个前缀 16 个地址）64。如果您运行的 Pod 多于可供分配的 IP 地址，则通告的容量与实际容量（剩余插槽）之间的这种不一致可能会导致问题。

话虽如此，你可以使用上述迁移策略将你的 Pod 从辅助 IP 地址安全地过渡到从前缀获取的地址。在两种模式之间切换时，Pod 将继续正常运行，并且：

- 从辅助 IP 模式切换到前缀委派模式时，分配给正在运行的 pod 的辅助 IP 地址不会被释放。前缀将分配给空闲插槽。Pod 终止后，它使用的辅助 IP 和插槽将被释放。
- 从前缀委托模式切换到辅助 IP 模式时，当该范围内的所有 IP 都不再分配给 pod 时，将释放前缀。如果将前缀中的任何 IP 分配给 pod，则该前缀将一直保留，直到 pod 终止。

### 前缀委派的调试问题

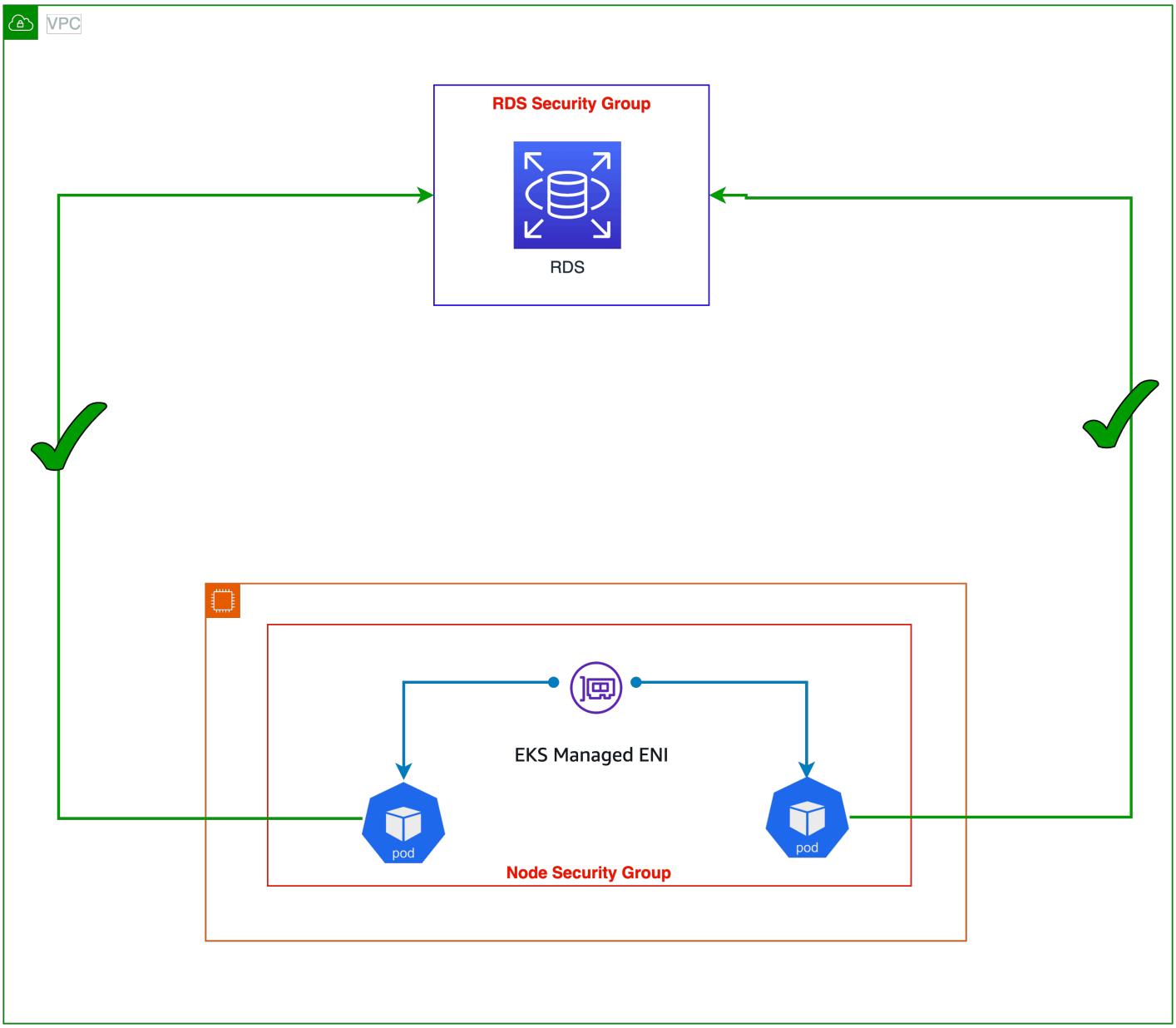
你可以在这里使用我们的调试指南来深入探讨你在Windows上前缀委托所面临的问题。

# 每个 Pod 的安全组

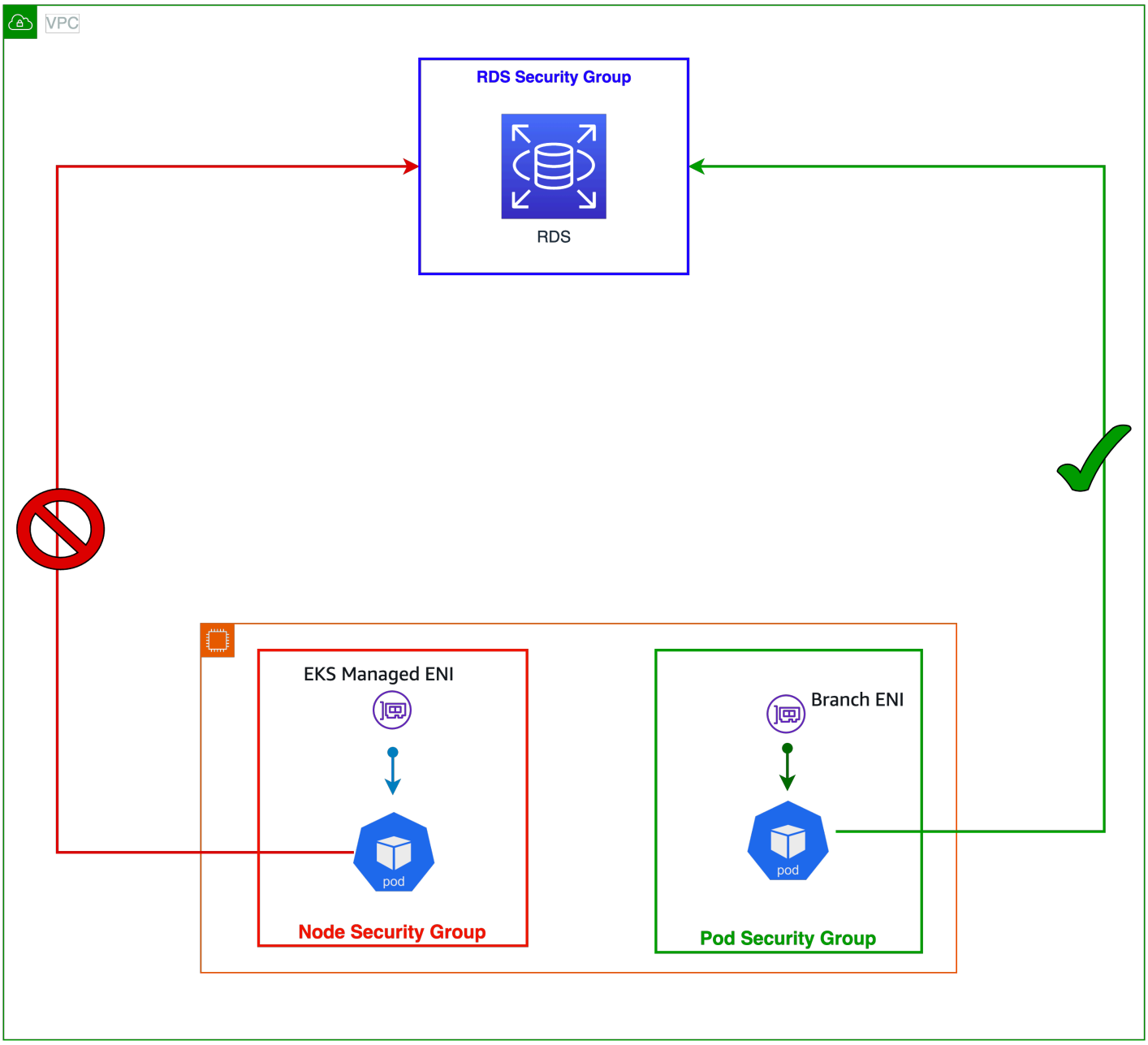
> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

AWS 安全组充当 EC2 实例的虚拟防火墙，用于控制入站和出站流量。默认情况下，Amazon VPC CNI 将使用与节点上的主 ENI 关联的安全组。更具体地说，与实例关联的每个 ENI 都将具有相同的 EC2 安全组。因此，节点上的每个 Pod 都与其运行所在的节点共享相同的安全组。

如下图所示，在工作节点上运行的所有应用程序 Pod 都将有权访问 RDS 数据库服务（考虑到 RDS 入站允许节点安全组）。安全组太粗糙了，因为它们适用于节点上运行的所有 Pod。Pods 安全组为工作负载提供网络分段，这是良好的深度防御策略的重要组成部分。

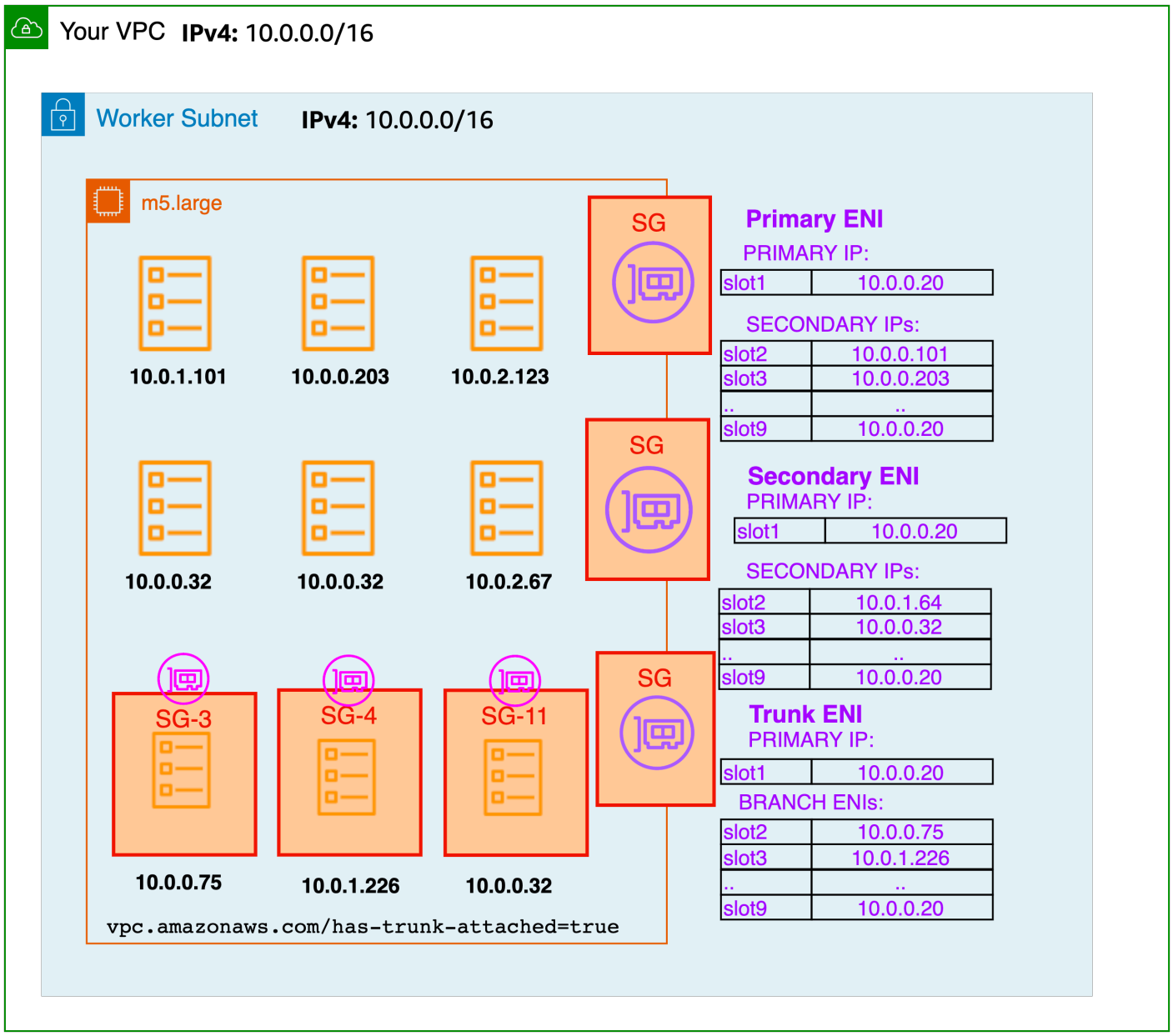



借助 Pod 的安全组，您可以通过在共享计算资源上运行具有不同网络安全要求的应用程序来提高计算效率。可以使用 EC2 安全组在一个地方定义多种类型的安全规则，例如 Pod-to-Pod 和 Pod-to-External AWS 服务，并将其应用于使用 Kubernetes 原生 API 的工作负载。下图显示了在 Pod 级别应用的安全组，以及它们如何简化您的应用程序部署和节点架构。Pod 现在可以访问 Amazon RDS 数据库了。




您可以通过设置 VPC CNI 来为 ENABLE_POD_ENI=true Pod 启用安全组。启用后，在控制平面上运行的 VPC 资源控制器（由 EKS 管理）会创建名为 “`aws-k8s-trunk-eni” 的中继接口并将其连接到节点。中继接口充当连接到实例的标准网络接口。要管理中继接口，您必须将AmazonEKSVPCResourceController托管策略添加到 Amazon EKS 集群附带的集群角色中。

控制器还会创建名为 “aws-k8s-branch-eni” 的分支接口，并将它们与中继接口关联起来。使用SecurityGroupPolicy自定义资源为 Pod 分配了一个安全组，并与分支接口关联。由于安全组是通过网络接口指定的，因此我们现在可以在这些额外的网络接口上安排需要特定安全组的 Pod。查看 EKS 用户指南中关于 Pod 安全组的部分，包括部署先决条件。




分支接口容量是对辅助 IP 地址的现有实例类型限制的补充。使用安全组的 Pod 不包含在 max-pods 公式中，当你对 Pod 使用安全组时，你需要考虑提高 max-pods 值，或者可以运行的容量少于节点实际支持的数量。

m5.large 最多可以有 9 个分支网络接口和最多 27 个辅助 IP 地址分配给其标准网络接口。如下例所示，m5.large 的默认最大容量为 29，EKS 将使用安全组的 Pod 计入最大 Pod 数。有关如何更改节点的最大 pod 数的说明，请参阅 E KS 用户指南。

当 Pod 的安全组与自定义网络结合使用时，将使用 Pod 的安全组中定义的安全组，而不是 eniconFig 中指定的安全组。因此，启用自定义联网后，在使用每个 Pod 的安全组时，请仔细评估安全组的顺序。

## 建议

### 禁用 TCP 早期解复用以进行存活探测

如果你使用的是活跃或就绪探针，你还需要禁用 TCP 早期解复用，这样 kubelet 就可以通过 TCP 连接到分支网络接口上的 Pod。只有在严格模式下才需要这样做。要执行此操作，请运行以下命令：

```
kubectl edit daemonset aws-node -n kube-system
```

在该`initContainer`部分下，将的值更改为 `DISABLE_TCP_EARLY_DEMUX` `true`.

### 使用适用于 Pod 的安全组来利用现有的 AWS 配置投资。

通过安全组，可以更轻松地限制对 VPC 资源的网络访问，例如 RDS 数据库或 EC2 实例。每个 Pod 的安全组的一个明显优势是可以重复使用现有 AWS 安全组资源。如果您使用安全组作为网络防火墙来限制对您的 AWS 服务的访问，我们建议使用分支 ENI 将安全组应用于 Pod。如果您要将应用程序从 EC2 实例转移到 EKS，并使用安全组限制对其他 AWS 服务的访问，请考虑为 Pod 使用安全组。

### 配置 Pod 安全组强制模式

Amazon VPC CNI 插件版本 1.11 添加了一个名为`POD_SECURITY_GROUP_ENFORCING_MODE`（“强制模式”）的新设置。强制模式既控制哪些安全组适用于 pod，也控制源 NAT 是否已启用。您可以将强制模式指定为严格模式或标准模式。默认为 “严格”，反映了`ENABLE_POD_ENI`设置为的 VPC CNI 之前的行为。`true`

在严格模式下，仅强制使用分支 ENI 安全组。源 NAT 也被禁用。

在标准模式下，将应用与主 ENI 和分支 ENI（与 Pod 关联）关联的安全组。网络流量必须符合两个安全组的要求。

> **Warning**
>
> 任何模式更改都只会影响新推出的 Pod。现有 Pod 将使用创建 Pod 时配置的模式。如果客户想要改变流量行为，则需要使用安全组回收现有 Pod。

### 强制模式：使用严格模式隔离 Pod 和节点流量：

默认情况下，Pod 的安全组设置为 “严格模式”。如果您必须将 Pod 流量与节点的其余流量完全分开，请使用此设置。在严格模式下，源 NAT 处于关闭状态，因此可以使用分支 ENI 出站安全组。

> **Warning**
>
> 启用严格模式后，所有来自 Pod 的出站流量都将离开节点并进入 VPC 网络。同一节点上 Pod 之间的流量将通过 VPC 传输。这会增加 VPC 流量并限制基于节点的功能。严格模式不支持 NodeLocal dnsCache。

### 强制模式：在以下情况下使用标准模式

Pod 中的容器可以看到客户端源 IP

如果您需要保持 Pod 中的容器可见客户端源 IP，请考虑`POD_SECURITY_GROUP_ENFORCING_MODE`将其设置为`standard`。Kubernetes 服务支持外部 TrafficPolicy =local，以支持保留客户端源 IP（默认类型的集群）。现在，你可以在标准模式下运行类型为 Kubernetes 的服务， NodePort 并 LoadBalancer 使用外部TrafficPolicy 设置为 “本地” 的实例目标。 `Local`保留客户端源 IP，避免再次跳转 LoadBalancer 并 NodePort 键入服务。

部署 NodeLocal dnsCache

对容器使用安全组时，请将标准模式配置为支持使用 NodeLocal DNSC ache 的 Pod。 NodeLocal dnsCache 通过在群集节点上运行 DNS 缓存代理来提高集群 DNS 性能。 DaemonSet这将帮助具有最高 DNS QPS 要求的 pod 查询本地 kube，dns/CoreDNS 拥有本地缓存，这将改善延迟。

NodeLocal 严格模式下不支持 DNSCache，因为所有网络流量（包括节点流量）都会进入 VPC。

支持 Kubernetes 网络政策

对于关联安全组的 Pod 使用网络策略时，我们建议使用标准强制模式。

我们强烈建议使用适用于 Pod 的安全组来限制对不属于集群的 AWS 服务的网络级访问。考虑使用网络策略来限制集群内 Pod 之间的网络流量，通常称为 East/West 流量。

### 识别每个 Pod 与安全组的不兼容性

Windows-based 而且非 nitro 实例不支持 Pod 的安全组。要在 Pod 中使用安全组，必须将实例标记为 is TrunkingEnabled。如果您的 Pod 不依赖于 VPC 内外的任何 AWS 服务，请使用网络策略来管理 Pod 之间的访问权限，而不是安全组。

## 使用每个 Pod 的安全组来有效地控制流向 AWS 服务的流量

如果在 EKS 集群中运行的应用程序必须与 VPC 内的其他资源（例如 RDS 数据库）通信，则可以考虑对 Pod 使用 SG。虽然有些策略引擎允许您指定 CIDR 或 DNS 名称，但在与终端节点位于 VPC 中的 AWS 服务进行通信时，它们并不是最佳选择。

相比之下，Kubernetes 网络策略提供了一种控制集群内外的入口和出口流量的机制。如果您的应用程序对其他 AWS 服务的依赖程度有限，则应考虑 Kubernetes 网络策略。您可以配置网络策略，根据 CIDR 范围指定出口规则，以限制对 AWS 服务的访问，而不是像 SG 这样的 AWS 原生语义。你可以使用 Kubernetes 网络策略来控制 Pod 之间（通常称为流量）以及 Pod 与外部服务之间的网络 East/West 流量。Kubernetes 网络策略是在 OSI 第 3 级和第 4 级实施的。

Amazon EKS 允许你使用 C alico 和 C ilium 等网络策略引擎。默认情况下，不安装网络策略引擎。有关如何设置的说明，请查看相应的安装指南。有关如何使用网络策略的更多信息，请参阅 EKS 安全最佳实践。DNS 主机名功能可在企业版的网络策略引擎中使用，该功能可用于控制 Kubernetes 与在 AWS 之外运行的资源 Services/Pods 之间的流量。此外，您可以考虑为默认不支持安全组的 AWS 服务提供 DNS 主机名支持。

## 标记单个安全组以使用 AWS 负载均衡器控制器

当为一个 Pod 分配了多个安全组时，Amazon EKS 建议将单个安全组标记为`kubernetes.io/cluster/$name`共享或所有安全组。该标签允许 AWS 负载均衡器控制器更新安全组规则，将流量路由到 Pod。如果只为 Pod 分配了一个安全组，则标签的分配是可选的。在安全组中设置的权限是累加的，因此标记单个安全组就足以让负载均衡器控制器找到和协调规则。它还有助于遵守安全组定义的默认配额。

## 为出站流量配置 NAT

对于来自分配了安全组的 Pod 的出站流量，将禁用源 NAT。对于使用需要访问互联网的安全组的 Pod，在配置有 NAT 网关或实例的私有子网上启动工作节点，并在 CNI 中启用外部 SN AT。

```
kubectl set env daemonset -n kube-system aws-node AWS_VPC_K8S_CNI_EXTERNALSNAT=true
```

## 将带有安全组的 Pod 部署到私有子网

分配了安全组的 Pod 必须在部署到私有子网的节点上运行。请注意，已分配安全组部署到公共子网的 Pod 将无法访问互联网。

## Verify 终止 GracePeriodSeconds 在 Pod 规范文件中

确保`terminationGracePeriodSeconds`在 Pod 规范文件中该值为非零（默认为 30 秒）。要让 Amazon VPC CNI 从工作节点中删除 Pod 网络，这是必不可少的。设置为零时，CNI 插件不会从主机上移除 Pod 网络，也无法有效清理分支 ENI。

## 对带有 Fargate 的 Pod 使用安全组

在 Fargate 上运行的 Pod 的安全组的工作方式与在 EC2 工作节点上运行的 Pod 非常相似。例如，在与 Fargate Pod 关联的 Fargate Pod 中引用安全组之前， SecurityGroupPolicy 您必须先创建安全组。默认情况下，如果您没有明确分配给 Fargate 容器，则会将集群安全组分配给 SecurityGroupPolicy 所有 Fargate Pod。为简单起见，你可能需要将集群安全组添加到 Fagate Pod 中， SecurityGroupPolicy 否则你必须向你的安全组添加最低安全组规则。你可以使用 describe-cluster API 来查找集群安全组。

```
 aws eks describe-cluster --name CLUSTER_NAME --query
 'cluster.resourcesVpcConfig.clusterSecurityGroupId'
```

```
cat >my-fargate-sg-policy.yaml <<EOF
apiVersion: vpcresources.k8s.aws/v1beta1
kind: SecurityGroupPolicy
metadata:
  name: my-fargate-sg-policy
  namespace: my-fargate-namespace
spec:
  podSelector:
    matchLabels:
      role: my-fargate-role
  securityGroups:
    groupIds:
      - cluster_security_group_id
      - my_fargate_pod_security_group_id
EOF
```

此处列出了最低安全组规则。这些规则允许 Fargate Pod 与 kube-apiserver、kubelet 和 CoreDNS 等集群内服务进行通信。您还需要添加规则以允许进出您的 Fargate Pod 的入站和出站连接。这将允许您的 Pod 与您的 VPC 中的其他 Pod 或资源进行通信。此外，您还必须包括规则，让 Fargate 从 Amazon ECR 或其他容器注册表（例如）提取容器镜像。 DockerHub有关更多信息，请参阅 AWS 一般参考中的 AWS IP 地址范围。

您可以使用以下命令来查找应用于 Fargate Pod 的安全组。

```
kubectl get pod FARGATE_POD -o jsonpath='{.metadata.annotations.fargate\.amazonaws
\.com/pod-sg}{"\n"}'
```

记下上面命令中的 eniID。

```
aws ec2 describe-network-interfaces --network-interface-ids ENI_ID --query
 'NetworkInterfaces[*].Groups[*]'
```

必须删除并重新创建现有的 Fargate 容器才能应用新的安全组。例如，以下命令启动示例应用程序的部署。要更新特定的 pod，可以在以下命令中更改命名空间和部署名称。

```
kubectl rollout restart -n example-ns deployment example-pod
```

# 负载均衡

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

负载均衡器接收传入流量，并将其分配到 EKS 集群中托管的预期应用程序的目标之间。这提高了应用程序的弹性。部署在 EKS 集群中时，AWS Load Balancer 控制器将为该集群创建和管理 AWS 弹性负载均衡器。创建类型LoadBalancer 为 Kubernetes 的服务时，AWS 负载均衡器控制器会创建一个网络负载均衡器 (NLB)，用于对 OSI 模型第 4 层接收到的流量进行负载均衡。而在创建 Kubernetes Ingress 对象时，AWS Load Balancer Contro ller 会创建一个应用程序负载均衡器 (ALB)，用于对 OSI 模型第 7 层的流量进行负载均衡。

## 选择 Load Balancer 类型

AWS Elastic Load Balancing (ELB) 产品组合支持以下负载均衡器：应用程序负载均衡器 (ALB)、网络负载均衡器 (NLB)、网关负载均衡器 (GWLB) 和经典负载均衡器 (CLB)。本最佳实践部分将重点介绍 ALB 和 NLB，这两者与 EKS 集群最为相关。

选择负载均衡器类型的主要考虑因素是工作负载要求。

有关更多详细信息以及所有 AWS 负载均衡器的参考，请参阅产品比较

### 如果您的工作负载为，请选择 Application Load Balancer (ALB) HTTP/HTTPS

如果工作负载需要在 OSI 模型的第 7 层进行负载平衡，则可以使用 AWS Load Balancer Controller 来配置 ALB；我们将在下一节中介绍配置。ALB 由前面提到的 Ingress 资源控制和配置，将 HTTP 或 HTTPS 流量路由到集群中的不同 Pod。ALB 为客户提供了更改应用程序流量路由算法的灵活性；默认路由算法是循环算法，而未完成请求最少的路由算法也是另一种选择。

### 如果您的工作负载是 TCP，或者您的工作负载需要保留客户端的源 IP，请选择 Network Load Balancer (NLB)

Network Load Balancer 在开放系统互联 (OSI) 模型的第四层（传输）上运行。它适用于基于 TCP 和 UDP 的工作负载。默认情况下，Network Load Balancer 在向容器呈现流量时还会保留客户端地址的源 IP。

### 如果您的工作负载无法使用 DNS，请选择 Network Load Balancer (NLB)

使用 NLB 的另一个关键原因是您的客户无法使用 DNS。在这种情况下，NLB 可能更适合您的工作负载，因为 Network Load Balancer 上的 IP 是静态的。虽然建议客户端在连接到负载均衡器时使用 DNS 将域名解析为 IP 地址，但如果客户端的应用程序不支持 DNS 解析并且只接受硬编码的 IP，那么 NLB 更合适，因为 IP 是静态的，并且在 NLB 的生命周期内保持不变。

## 配置负载均衡器

在确定最适合您的工作负载的 Load Balancer 之后，客户可以选择多种配置负载均衡器。

### 通过部署 AWS Load Balancer 控制器来配置负载均衡器

在 EKS 集群中配置负载均衡器有两种主要方法。

- 利用 AWS 云提供商中的服务控制器（旧版）
- 利用 AWS Load Balancer 控制器（推荐）

默认情况下，Kubernetes 服务控制器（也称为树内控制器）会协调类型的 Kubernetes 服务资源。LoadBalancer此控制器内置在 AWS 云提供商组件中，该组件用作 Kubernetes 云控制器管理器。

已配置的 Elastic Load Balancer 的配置由必须添加到 Kubernetes 服务清单中的注释控制。服务控制器和 AWS Load Balancer 控制器使用的注解不同。

服务控制器是旧版，目前仅修复了严重的错误。当您创建类型为 Kubernetes 服务时 LoadBalancer，服务控制器会默认创建 AWS 负载均衡，但如果您使用正确的注释，也可以创建 AWS NLB。值得注意的是，服务控制器不支持 Kubernetes 入口资源，也不支持 IPv6。

我们建议在您的 EKS 集群中使用 AWS Load Balancer 控制器来协调 Kubernetes 服务和入口资源。您必须在 Kubernetes 服务或 Ingress 清单中使用正确的注释，这样 AWS Load Balancer Controller 才能拥有协调流程。（而不是服务控制器）

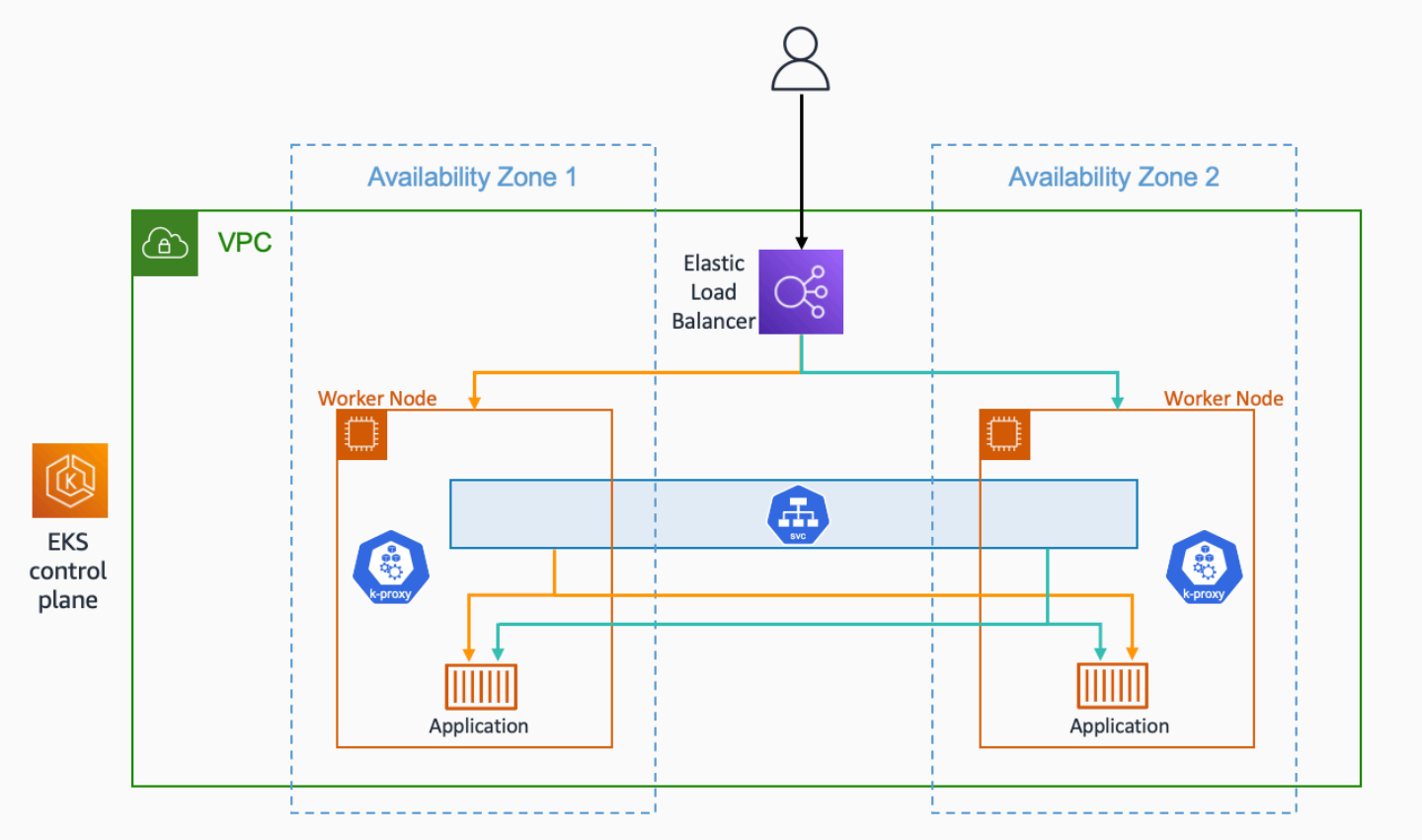
如果您使用的是 EKS 自动模式，则会自动为您提供 AWS Load Balancer 控制器；无需安装。

# 选择 Load Balancer Target-Type

## 使用 IP 将 Pod 注册为目标 Target-Type

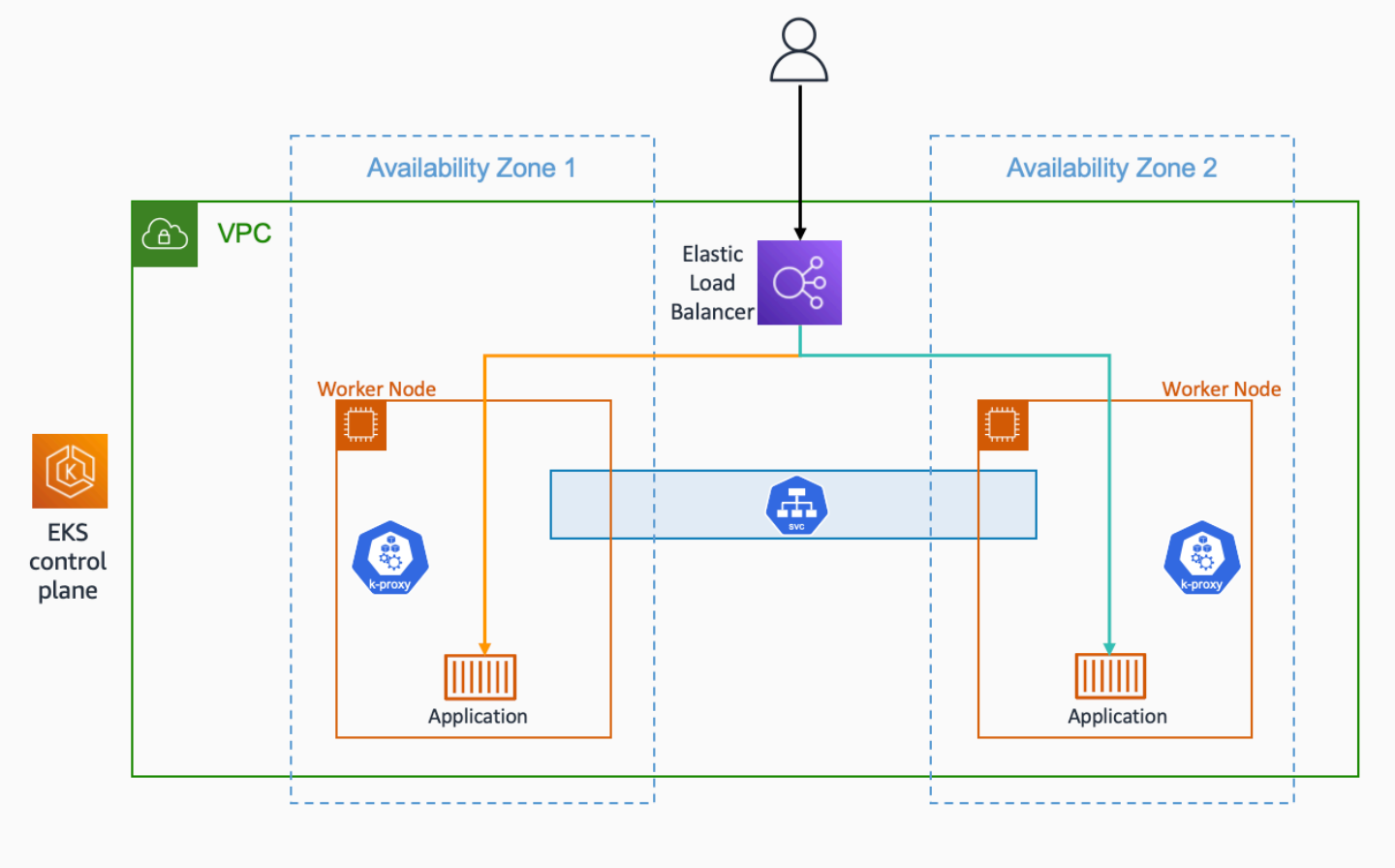
AWS Elastic Load Balancer：网络和应用程序，将收到的流量发送到目标组中的注册目标。对于 EKS 集群，您可以在目标组中注册两种类型的目标：实例和 IP，使用哪种目标类型会影响注册的内容以及流量从 Load Balancer 路由到 Pod 的方式。默认情况下，AWS Load Balancer 控制器将使用 “实例” 类型注册目标，该目标将是工作节点的 IPNodePort，其含义包括：

- 来自 Load Balancer 的流量将被转发到上的 Worker 节点 NodePort，这由 iptables 规则（由节点上运行的 kube-proxy 配置）处理，然后通过其 clusterIP（仍在节点上）转发到服务，最后，服务会随机选择一个注册到它的 pod 并将流量转发给它。此流程涉及多个跃点，可能会产生额外的延迟，特别是因为服务有时会选择在另一个工作节点上运行的 pod，而该工作节点也可能位于另一个可用区中。
- 由于 Load Balancer 将工作节点注册为其目标，这意味着发送到目标的运行状况检查不会直接被 Pod 接收，而是由 Pod 上的工作节点接收，运行状况检查流量将遵循上述相同的路径。 NodePort
- 监控和故障排除更为复杂，因为 Load Balancer 转发的流量不会直接发送到 pod，因此您必须仔细关联工作节点上收到的数据包与服务 ClusterIP，并最终关联到 Pod，才能完全端到端地了解数据包的路径，以便进行正确的故障排除。




相比之下，如果您按照我们的建议将目标类型配置为 “IP”，则含义将如下：

- 来自 Load Balancer 的流量将直接转发到 pod，这简化了网络路径，因为它绕过了之前额外的 Worker 节点和服务集群 IP 跳数，减少了服务将流量转发到另一个可用区中的 pod 时本来会产生的延迟，最后它消除了 iptables 规则在工作节点上的开销处理。
- 负载均衡器的运行状况检查由 Pod 直接接收和响应，这意味着目标状态 “健康” 或 “不健康” 直接表示 Pod 的健康状态。
- 监控和故障排除更容易，任何用于捕获数据包 IP 地址的工具都将在其源和目标字段中直接显示 Load Balancer 和 pod 之间的双向流量。




要创建使用 IP 目标的 AWS Elastic Load Balancing，请添加：

- `alb.ingress.kubernetes.io/target-type: ip`配置 Kubernetes Ingress（Application Load Balancer）时的 Ingress 清单的注释
- `service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip`在配置你的 Kubernetes 服务类型 LoadBalancer （网络负载均衡器）时，在你的服务清单中添加注释。

## 配置 Load Balancer 运行状况检查

虽然 Kubernetes 提供了自己的运行状况检查机制（将在下一节中详细介绍），但我们建议实施 ELB 运行状况检查作为在 Kubernetes 控制平面之外运行的补充保护措施。即使在以下情况下，这个独立层也能继续监视您的应用程序：

- Kubernetes 控制面板中断
- 探测器执行延迟
- kubelet 和 pod 之间的网络分区

对于在上述场景中需要最大可用性加速恢复的关键工作负载，ELB 运行状况检查提供了一个与 Kubernetes 原生机制配合使用而不是取代 Kubernetes 原生机制的基本安全网。

要在 ELB 上配置和微调运行状况检查，您必须在 Kubernetes 服务或 Ingress 清单中使用将由服务控制器或 AWS Load Balancer Controller 协调的注释。

## 可用性和 Pod 生命周期

在应用程序升级期间，您必须确保您的应用程序始终可用于处理请求，这样用户就不会遇到任何停机时间。在这种情况下，一个常见的挑战是在 Kubernetes 层和基础架构（例如外部负载均衡器）之间同步工作负载的可用性状态。接下来的几节将重点介绍解决此类情况的最佳实践。

> **Note**
>
> 以下解释是基于的，EndpointSlices因为它是 Kubernetes 中端点的推荐替代品。就下文所述情景而言，两者之间的差异可以忽略不计。默认情况下，AWS Load Balancer Controller 使用终端节点，您可以 EndpointSlices 通过在控制器上启用 en able-endpoint-sliceflag 来启用。

### 使用运行状况检查

默认情况下，Kubernetes 会运行进程运行状况检查，节点上的 kubelet 进程会验证容器的主进程是否正在运行。如果不是，则默认情况下它会重新启动该容器。但是，您也可以配置 Kubernetes 探测器来识别容器进程何时正在运行但处于死锁状态，或者应用程序是否成功启动。探测器可以基于 exec、grpc、HttpGet 和 tcpSocket 机制。根据探测的类型和结果，可以重新启动容器。

请参阅下方附录中的 Pod 创建部分，重新了解 Pod 创建过程中的事件顺序。

### 使用就绪探测器

默认情况下，当 Pod 中的所有容器都在运行时，Pod 条件被视为 “就绪”。但是，应用程序可能仍无法处理客户请求。例如，应用程序可能需要从外部资源提取一些数据或配置才能处理请求。在这种状态下，你既不想终止应用程序，也不想向其转发任何请求。Readiness p@@ rob e 使您能够确保 Pod 未被视为 “就绪”，这意味着在探测结果出来之前，Pod 不会被添加到 EndpointSlice对象中`success`。另一方面，如果探测器在更远的距离内失败，则会将 Pod 从 EndpointSlice 对象中移除。您可以在 Pod 清单中为每个容器配置就绪探测器。 `kubelet`每个节点上的进程对该节点上的容器运行就绪探测。

## 利用 Pod 就绪门

就绪探测器的一个方面是它没有外部feedback/influence 机制，节点上的 kubelet 进程执行探测器并定义探测器的状态。这不会对 Kubernetes 层中微服务本身之间的请求（东西向流量）产生任何影响，因为 EndpointSlice 控制器会使端点列表 (Pod) 始终保持最新。那么，为什么以及何时需要外部机制呢？

当您使用 Load Balancer 或 Kubernetes Ingress（用于南北流量）的 Kubernetes 服务类型公开应用程序时，必须将相应的 Kubernetes 服务的 Pod IP 列表传播到外部基础设施负载均衡器，以便负载均衡器也具有最新的列表目标。AWS Load Balancer Contro ller 在这里弥合了差距 当你使用 AWS Load Balancer Controller 并利用`target group: IP`时，就像 `kube-proxy` AWS Load Balancer 控制器也会收到更新（通过`watch`），然后它会与 ELB API 通信，配置并开始在 ELB 上将 Pod IP 注册为目标。

当你对部署进行滚动更新时，会创建新的 Pod，一旦新 Pod 的状态为 “就绪”， old/existing Pod 就会被终止。在此过程中，Kubernetes EndpointSlice 对象的更新速度快于 ELB 将新 Pod 注册为目标所需的时间，请参阅目标注册。在短时间内，Kubernetes 层和可能丢弃客户端请求的基础设施层之间可能会出现状态不匹配。在这段时间内，在 Kubernetes 层中，新的 Pod 已经准备好处理请求了，但从 ELB 的角度来看，它们还没有。

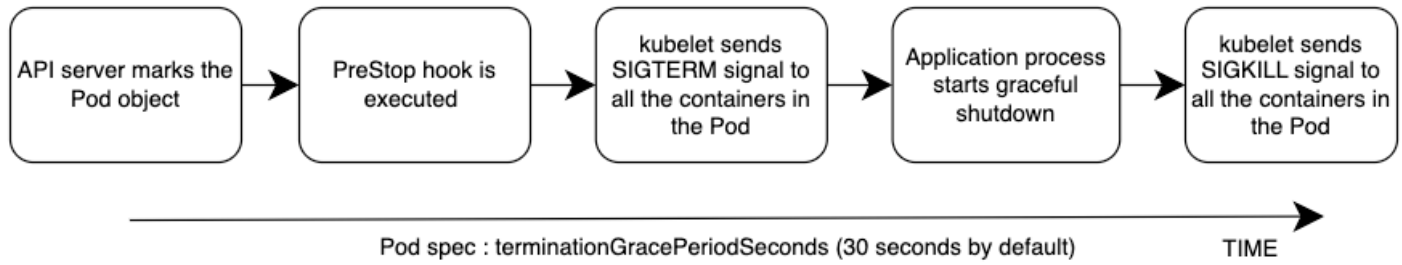
Pod Readiness Gates 允许您定义在 Pod 条件被视为 “就绪” 之前必须满足的其他要求。对于 AWS ELB，AWS Load Balancer 控制器监控 AWS ELB 上目标（Pod）的状态，一旦目标注册完成且其状态变为 “正常”，控制器就会将 Pod 的状态更新为 “就绪”。使用这种方法，您可以根据外部网络的状态（即 AWS ELB 上的目标状态）来影响 Pod 状况。Pod Readiness Gates 在滚动更新场景中至关重要，因为它使您能够防止部署的滚动更新终止旧 pod，直到 AWS ELB 上新创建的 Pod 目标状态变为 “正常”。

## 优雅地关闭应用程序

您的应用程序应通过开始正常关闭来响应 SIGTERM 信号，这样客户端就不会遇到任何停机时间。这意味着您的应用程序应该运行清理程序，例如保存数据、关闭文件描述符、关闭数据库连接、优雅地完成动态请求并及时退出以满足 Pod 终止请求。您应该将宽限期设置为足够长的时间，以便清理工作可以完成。要了解如何响应 SIGTERM 信号，您可以参考用于应用程序的相应编程语言的资源。

如果您的应用程序在收到 SIGTERM 信号后无法正常关闭，或者它ignores/does 没有收到信号，那么您可以改用 PreStophook 来启动应用程序的优雅关闭。Prestop hook 在发送 SIGTERM 信号之前立即执行，它可以执行任意操作，而不必在应用程序代码本身中实现这些操作。

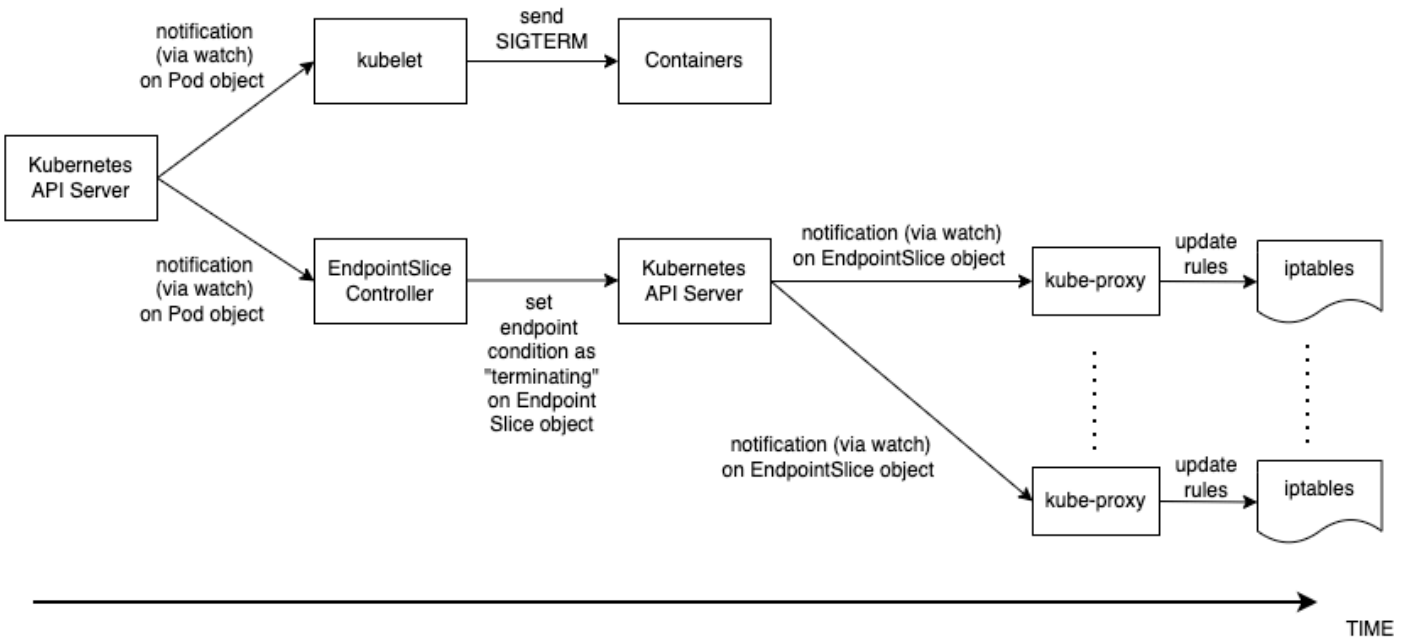
事件的总体顺序如下图所示。注意：无论应用程序的正常关闭过程的结果如何，或者 PreStop 挂钩的结果如何，应用程序容器最终都会通过 SIGKILL 在宽限期结束时终止。




请参阅下方附录中的 Pod 删除部分，重新了解 Pod 删除过程中的事件顺序。

## 优雅地处理客户请求

Pod 删除中的事件顺序与 Pod 创建中的事件顺序不同。创建 Pod 后，会`kubelet`更新 Kubernetes API 中的 Pod IP，然后才会更新 EndpointSlice 对象。另一方面，当 Pod 被终止时，Kubernetes API 会同时通知 kubelet 和 EndpointSlice 控制器。仔细检查下图，该图显示了事件的顺序。




状态从 API 服务器一直传播到上面解释的节点上的 iptables 规则的方式创造了一个有趣的竞争条件。因为容器很有可能比每个节点上的 kube-proxy 更早地收到 SIGKILL 信号，所以更新本地 iptables 规则。在这种情况下，值得一提的两种情况是：

- 如果您的应用程序在收到SIGTERM后立即直言不讳地丢弃了正在进行的请求和连接，这意味着客户端将看到50倍的错误。

- 即使您的应用程序确保在收到 SIGTERM 后完全处理所有正在进行的请求和连接，但在宽限期内，新的客户端请求仍会发送到应用程序容器，因为 iptables 规则可能仍未更新。在清理过程关闭容器上的服务器套接字之前，这些新请求将导致新的连接。当宽限期结束时，这些在 SIGTERM 之后建立的连接将在发送 SIGKILL 后无条件中断。

在 Pod 规范中设置足够长的宽限期可能会解决这个问题，但是根据传播延迟和实际客户端请求的数量，很难预测应用程序优雅地关闭连接所需的时间。因此，这里不太完美但最可行的方法是使用 PreStop 钩子将SIGTERM信号延迟到iptables规则更新之前，以确保没有新的客户端请求发送到应用程序，而是只有现有的连接才能继续。 PreStop hook 可以是一个简单的 Exec 处理程序，例如。`sleep 10`

当您使用 Kubernetes 服务类型 Load Balancer 或 Kubernetes Ingress（适用于南北流量）使用 AWS Load Balancer Controller 和杠杆来公开应用程序时，上述行为和建议同样适用。`target group: IP`因为就像 `kube-proxy` AWS Load Balancer Controller 也会收到 EndpointSlice 对象的更新（通过监视），然后它与 ELB API 通信以开始从 ELB 注销 Pod IP。但是，根据 Kubernetes API 或 ELB API 的负载，这也可能需要一些时间，而且 SIGTERM 可能很久以前就已经发送到应用程序了。ELB 开始注销目标后，它将停止向该目标发送请求，因此应用程序将不会收到任何新请求，并且 ELB 还会开始取消注册延迟，默认为 300 秒。在注销过程中，目标基本上是 ELB 等待与该目标的飞行中 requests/existing 连接耗尽`draining`的地方。取消注册延迟到期后，目标将处于未使用状态，并且向该目标发出的任何正在进行的请求都将被强制丢弃。

### 使用 Pod 中断预算

为您的应用程序配置 Pod 中断预算 (PDB)。pdbLimits 由于自愿中断而同时关闭的复制应用程序的 Pod 数量。它可确保或部署中保持最低数量或百分比的 pod 可用。 StatefulSet 例如，基于法定人数的应用程序需要确保运行的副本数量永远不会低于法定人数所需的数量。或者，Web 前端可以确保提供负载的副本数量永远不会低于总数的特定百分比。PDB 将保护应用程序免受诸如节点耗尽或推出新版本部署之类的操作的影响。请记住，PDB 无法保护应用程序免受非自愿中断的影响，例如节点操作系统故障或网络连接中断。有关更多信息，请参阅在 Kubernetes 中为您的应用程序指定中断预算文档。

## 参考

- KubeCon 2019 年欧洲会议—— 准备好了吗？ 深入探讨 Service Health 的 Pod 就绪之门
- 图书-Kubernetes 在行动
- AWS 博客-如何在 EKS 上使用 ALB 快速扩展您的应用程序（不会损失流量）

# 附录

## Pod 创建

在部署 Pod 然后开始 healthy/ready 接收和处理客户端请求的场景中，必须了解事件的顺序。让我们来谈谈事件的顺序。

1. Pod 是在 Kubernetes 控制平面上创建的（即通过 kubectl 命令、部署更新或扩展操作）。
2. `kube-scheduler`将 Pod 分配给集群中的一个节点。
3. 在分配的节点上运行的 kubelet 进程（通过`watch`）接收更新，并与容器运行时通信以启动 Pod 规范中定义的容器。
4. 当容器开始运行时，kubelet 会像 Kubernet es API Ready 中的 Pod 对象一样更新 Pod 条件。
5. EndpointSlice控制器接收 Pod 状态更新（通过`watch`），并将 Pod IP/Port 作为新端点添加到相应 Kubernetes 服务的EndpointSlice对象（Pod IP 列表）中。
6. 每个节点上的 kube-proxy 进程（通过`watch`）接收EndpointSlice 对象的更新，然后使用新的 Pod 更新每个节点上的 iptables 规则。 IP/port

## 吊舱删除

就像创建 Pod 一样，必须了解 Pod 删除期间的事件顺序。让我们来谈谈事件的顺序。

1. Pod 删除请求会发送到 Kubernetes API 服务器（即通过`kubectl`命令、部署更新或扩展操作）。
2. Kubernetes API 服务器通过在 P od 对象中设置 deletionTimeStamp 字段来启动宽限期，默认为 30 秒。（宽限期可以通过以下方式在 Pod 规范中配置`terminationGracePeriodSeconds`）
3. 节点上运行的`kubelet`进程接收 Pod 对象的更新（通过 watch），并向该 Pod 中每个容器内的进程标识符 1（PID 1）发送 SIGTER M 信号。然后它看`terminationGracePeriodSeconds`.
4. EndpointSlice控制器还接收步骤 2 中的更新（通过`watch`），并在相应的 Kubernetes 服务的EndpointSlice对象（Pod IP 列表）中将端点条件设置为 “终止”。
5. 每个节点上的 kube-proxy 进程（通过`watch`）接收EndpointSlice 对象的更新，然后 kube-proxy 更新每个节点上的 iptable s 规则，以停止将客户端的请求转发到 Pod。
6. 当`terminationGracePeriodSeconds`过期时，会向 Pod 中每个容器的父进程`kubelet`发送 SIGKI LL 信号并强制终止它们。
7. TheEndpointSlice控制器从EndpointSlice对象中移除端点。
8. API 服务器会删除 Pod 对象。

# 监控 EKS 工作负载是否存在网络性能问题

## 监控 CoreDNS 流量是否存在 DNS 限制问题

由于 DNS 限制，运行 DNS 密集型工作负载有时会出现间歇性的 CoreDNS 故障，这可能会影响可能偶尔遇到错误的应用程序。 UnknownHostException

CoreDNS 部署具有反关联性策略，指示 Kubernetes 调度器在集群中的不同工作节点上运行 CoreDNS 实例，也就是说，它应避免将副本放在同一个工作节点上。这有效地减少了每个网络接口的 DNS 查询数量，因为来自每个副本的流量都通过不同的 ENI 路由。如果您注意到 DNS 查询由于每秒 1024 个数据包的限制而受到限制，则可以 1) 尝试增加 CoreDNS 副本的数量或 2) 实现 dnsCache。NodeLocal 有关更多信息，请参阅监控 CoreDNS 指标。

### Challenge

- 丢包会在几秒钟内发生，要正确监控这些模式以确定 DNS 限制是否真的在发生，对我们来说可能很棘手。
- DNS 查询在 elastic network interface 级别受到限制。因此，受限制的查询不会出现在查询日志中。
- 流日志不会捕获所有 IP 流量。 E.g。实例与 Amazon DNS 服务器联系时生成的流量。如果您使用自己的 DNS 服务器，则会记录到该 DNS 服务器的所有流量

### 解决方案

识别工作节点中 DNS 限制问题的一种简单方法是捕`linklocal_allowance_exceeded`获指标。linklocal_allowance_e xceed 是由于流向本地代理服务的流量的 PPS 超过网络接口的最大值而丢弃的数据包数。这会影响流向 DNS 服务、实例元数据服务和 Amazon Time Sync Service 的流量。与其实时跟踪此事件，我们也可以将该指标流式传输到适用于 Prometheus 的亚马逊托管服务，并可以在亚马逊托管 Grafana 中对其进行可视化

## 使用 Conntrack 指标监控 DNS 查询延迟

另一个可以帮助监控 CoreDNS 限制/查询延迟的指标是和。`conntrack_allowance_available` `conntrack_allowance_exceeded`因超过 Connections Tracked 限额而导致的连接故障可能比因超出其他限额而造成的影响更大。在依赖 TCP 传输数据时，由于 TCP 的拥塞控制功能，由于超过 EC2 实例网络限额而排队或丢弃的数据包（例如带宽、PPS 等）通常可以得到优雅的处理。受影响的流量将减慢，丢失的数据包将被重新传输。但是，当实例超出其连接跟踪限额时，只有关闭一些现有连接以便为新连接腾出空间，才能建立新的连接。

conntrack_allowance_available并conntrack_allowance_exceeded帮助客户监控跟踪的连接限额，该限额因每个实例而异。这些网络性能指标使客户能够了解当实例的网络限额（例如网络带宽 Packets-Per-Second (PPS)、跟踪的连接和 Link-local 服务访问权限（Amazon DNS、实例元数据服务、Amazon Time Sync）超出时排队或丢弃的数据包数量

conntrack_allowance_available是指在达到该实例类型的 “已跟踪连接” 限额之前，该实例可以建立的跟踪连接数（仅支持基于 nitro 的实例）。 conntrack_allowance_exceeded是由于连接跟踪超过实例的最大值且无法建立新连接而丢弃的数据包数。

## 其他重要的网络性能指标

其他重要的网络性能指标包括：

bw_in_allowance_exceeded（指标的理想值应为零）是由于入站聚合带宽超过实例的最大值而排队 and/or 丢弃的数据包数

bw_out_allowance_exceeded（指标的理想值应为零）是由于出站聚合带宽超过实例的最大值而排队 and/or 丢弃的数据包数

pps_allowance_exceeded（指标的理想值应为零）是由于双向 PPS 超过实例的最大值而排队 and/or 丢弃的数据包数

## 捕获指标以监控工作负载是否存在网络性能问题

弹性网络适配器 (ENA) 驱动程序从启用弹性网络适配器 (ENA) 的实例中发布上述网络性能指标。所有网络性能指标都可以使用 CloudWatch 代理发布到 CloudWatch 。有关更多信息，请参阅博客。

现在让我们捕捉上面讨论的指标，将其存储在适用于 Prometheus 的亚马逊托管服务中，然后使用亚马逊托管 Grafana 进行可视化

### 先决条件

- ethtool-确保工作节点安装了 ethtool
- 在您的 AWS 账户中配置的 AMP 工作空间。有关说明，请参阅 AMP 用户指南中的创建工作空间。
- 亚马逊托管 Grafana 工作区

## 部署 Prometheus Node Exporter

为了收集工作节点的 ethtool 指标，我们需要部署一个收集器来收集指标并以 Prometheus 格式公开这些指标。以下命令将添加 `prometheus-community` Helm 存储库并在启用`ethtool`收集器的情况下安装`prometheus-node-exporter`图表。

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm upgrade -i prometheus-node-exporter prometheus-community/prometheus-node-exporter
 \
  --set extraArgs[0]="--collector.ethtool" \
  --set extraArgs[1]="--collector.ethtool.device-include=(eth|em|eno|ens|enp)[0-9s]+" \
  --set extraArgs[2]="--collector.ethtool.metrics-include=.*" \
  --set podAnnotations."prometheus\.io/scrape='true'" \
  --set podAnnotations."prometheus\.io/port='9100'"
```

## 部署 ADOT 收集器来抓取 ethtool 指标并存储在适用于 Prometheus 的亚马逊托管服务工作空间中

您安装了 AWS Distro OpenTelemetry (ADOT) 的每个集群都必须具有此角色才能向您的 AWS 服务账户授予将指标存储到适用于 Prometheus 的亚马逊托管服务的权限。按照以下步骤创建 IAM 角色并使用 IRSA 将其与 Amazon EKS 服务账户关联：

```
eksctl create iamserviceaccount --name adot-collector --namespace default
 --cluster <CLUSTER_NAME> --attach-policy-arn arn:aws:iam::aws:policy/
AmazonPrometheusRemoteWriteAccess --attach-policy-arn arn:aws:iam::aws:policy/
AWSXrayWriteOnlyAccess --attach-policy-arn arn:aws:iam::aws:policy/
CloudWatchAgentServerPolicy --region <REGION> --approve  --override-existing-
serviceaccounts
```

让我们部署 ADOT 收集器从 prometheus ethool 导出器中获取指标，然后将其存储在 Prometheus 的亚马逊托管服务中

以下过程使用以部署为模式值的 YAML 文件示例。这是默认模式，与独立应用程序类似地部署 ADOT 收集器。此配置接收来自示例应用程序的 OTLP 指标和从集群上的 Pod 中抓取的 Amazon Prometheus 托管服务指标

```
curl -o collector-config-amp.yaml https://raw.githubusercontent.com/aws-observability/
aws-otel-community/master/sample-configs/operator/collector-config-amp.yaml
```

在 collector-config-amp.yaml 中，用你自己的值替换以下内容：

- 模式：部署
- 服务账户：dot-collectol
- 端点：<YOUR_REMOTE_WRITE_ENDPOINT>
- 区域:<YOUR_AWS_REGION>
- 名称：adot-collector

```
kubectl apply -f collector-config-amp.yaml
```

部署 adot 收集器后，这些指标将成功存储在 Amazon Prometheus 中

## 在亚马逊托管服务中配置警报管理器，让 Prometheus 发送通知

您可以使用适用于 Prometheus 的亚马逊托管服务中的警报管理器为关键警报设置警报规则，然后可以向亚马逊 SNS 主题发送通知。让我们配置记录规则和警报规则，以检查到目前为止讨论的指标。

我们将使用适用于 Prometheus 的亚马逊托管服务的 ACK 控制器来配置警报和记录规则。

让我们为适用于 Prometheus 的亚马逊托管服务 Prometheus 服务部署 ACL 控制器：

```
export SERVICE=prometheusservice
export RELEASE_VERSION=`curl -sL https://api.github.com/repos/aws-controllers-k8s/
$SERVICE-controller/releases/latest | grep '"tag_name":' | cut -d'"' -f4`
export ACK_SYSTEM_NAMESPACE=ack-system
export AWS_REGION=us-east-1
aws ecr-public get-login-password --region us-east-1 | helm registry login --username
 AWS --password-stdin public.ecr.aws
helm install --create-namespace -n $ACK_SYSTEM_NAMESPACE ack-$SERVICE-controller \
oci://public.ecr.aws/aws-controllers-k8s/$SERVICE-chart --version=$RELEASE_VERSION --
set=aws.region=$AWS_REGION
```

运行该命令，片刻之后你应该会看到以下消息：

```
You are now able to create Amazon Managed Service for Prometheus (AMP) resources!

The controller is running in "cluster" mode.

The controller is configured to manage AWS resources in region: "us-east-1"
```

```
The ACK controller has been successfully installed and ACK can now be used to provision
 an Amazon Managed Service for Prometheus workspace.
```

现在让我们创建一个用于配置警报管理器定义和规则组的 yaml 文件。将以下文件另存为 `rulegroup.yaml`

```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: RuleGroupsNamespace
metadata:
   name: default-rule
spec:
   workspaceID: <Your WORKSPACE-ID>
   name: default-rule
   configuration: |
     groups:
     - name: ppsallowance
       rules:
       - record: metric:pps_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}[30s])
       - alert: PPSAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical

         annotations:
           summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
           description: "PPSAllowanceExceeded is greater than 0"
     - name: bw_in
       rules:
       - record: metric:bw_in_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
[30s])
       - alert: BWINAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical

         annotations:
           summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
```

```
            description: "BWInAllowanceExceeded is greater than 0"
      - name: bw_out
        rules:
        - record: metric:bw_out_allowance_exceeded
          expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
[30s])
        - alert: BWOutAllowanceExceeded
          expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
 [30s]) > 0
          labels:
            severity: critical

          annotations:
            summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
            description: "BWoutAllowanceExceeded is greater than 0"
      - name: conntrack
        rules:
        - record: metric:conntrack_allowance_exceeded
          expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
[30s])
        - alert: ConntrackAllowanceExceeded
          expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
 [30s]) > 0
          labels:
            severity: critical

          annotations:
            summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
            description: "ConnTrackAllowanceExceeded is greater than 0"
      - name: linklocal
        rules:
        - record: metric:linklocal_allowance_exceeded
          expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
[30s])
        - alert: LinkLocalAllowanceExceeded
          expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
 [30s]) > 0
          labels:
            severity: critical

          annotations:
```

```
          summary: Packets dropped due to PPS rate allowance exceeded for local
 services (instance {{ $labels.instance }})
          description: "LinkLocalAllowanceExceeded is greater than 0"
```

将您的 WORKSPACE-ID ID 替换为您正在使用的工作区的工作区 ID。

现在让我们配置警报管理器定义。将以下文件另存为 `alertmanager.yaml`

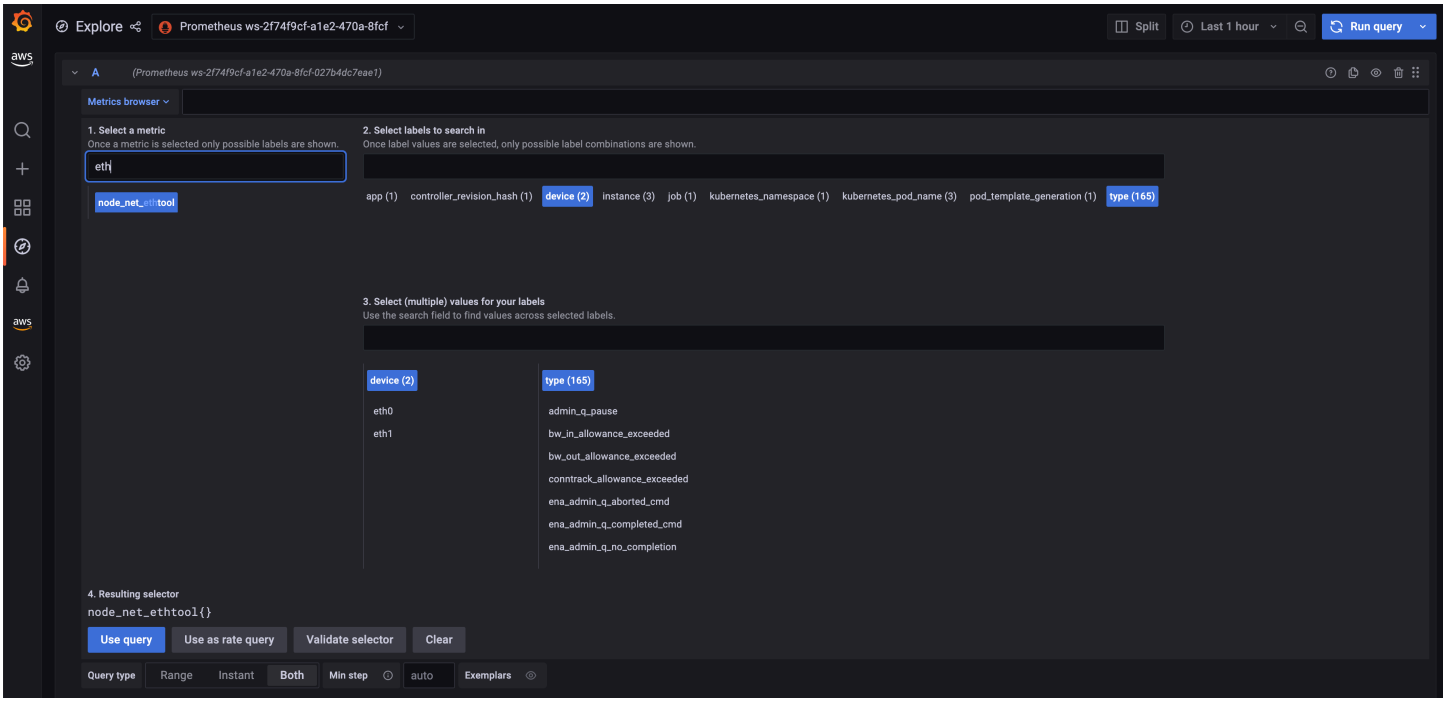
```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: AlertManagerDefinition
metadata:
  name: alert-manager
spec:
  workspaceID: <Your WORKSPACE-ID >
  configuration: |
    alertmanager_config: |
      route:
         receiver: default_receiver
       receivers:
       - name: default_receiver
          sns_configs:
          - topic_arn: TOPIC-ARN
            sigv4:
              region: REGION
            message: |
              alert_type: {{ .CommonLabels.alertname }}
              event_type: {{ .CommonLabels.event_type }}
```

将您 WORKSPACE-ID 替换为新工作空间的工作空间 ID，将您 TOPIC-ARN 替换为要发送警报的 Amazon 简单通知服务主题的 ARN，将区域替换为工作负载的当前区域。请确保您的工作空间有权向 Amazon SNS 发送消息。

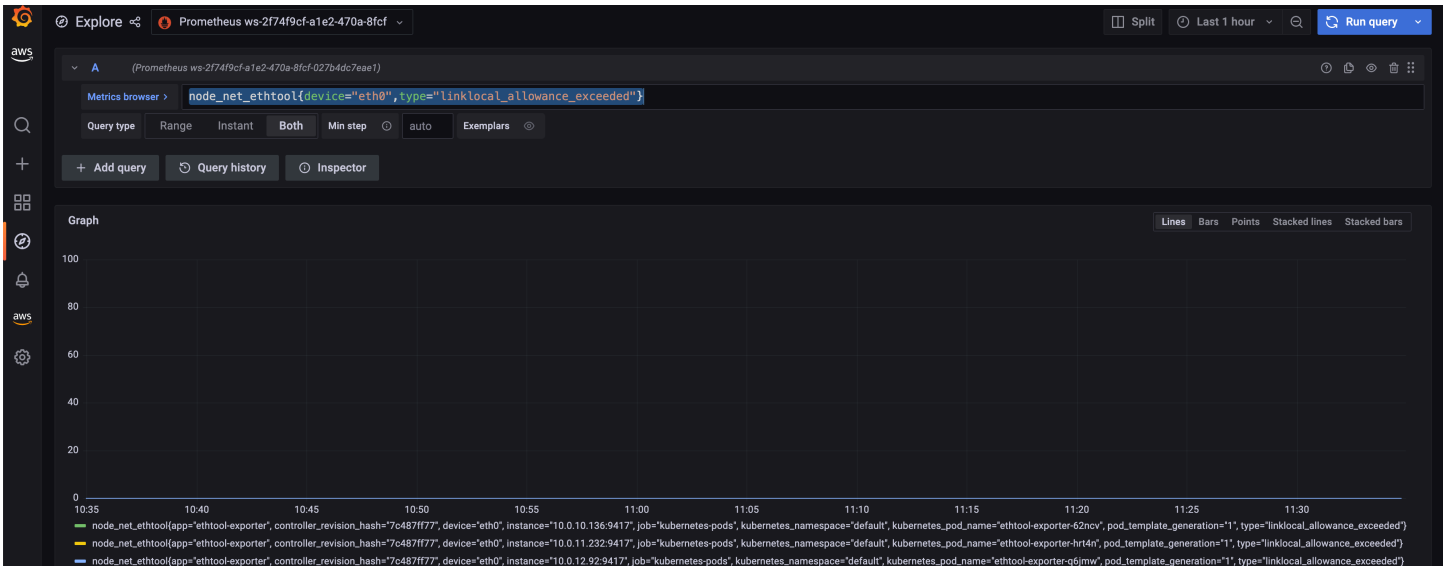
## 在 Amazon Managed Grafana 中可视化 ethtool 指标

让我们来可视化亚马逊托管 Grafana 中的指标并创建一个控制面板。将适用于 Prometheus 的亚马逊托管服务配置为亚马逊托管 Grafana 控制台内的数据源。有关说明，请参阅添加亚马逊 Prometheus 作为数据源

现在让我们来探索 Amazon Managed Grafana 中的指标：点击 “浏览” 按钮，然后搜索 ethtool：




让我们使用查询为 linklocal_allowance_exceed 指标构建一个仪表板。`rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}[30s])`它将生成下面的仪表板。

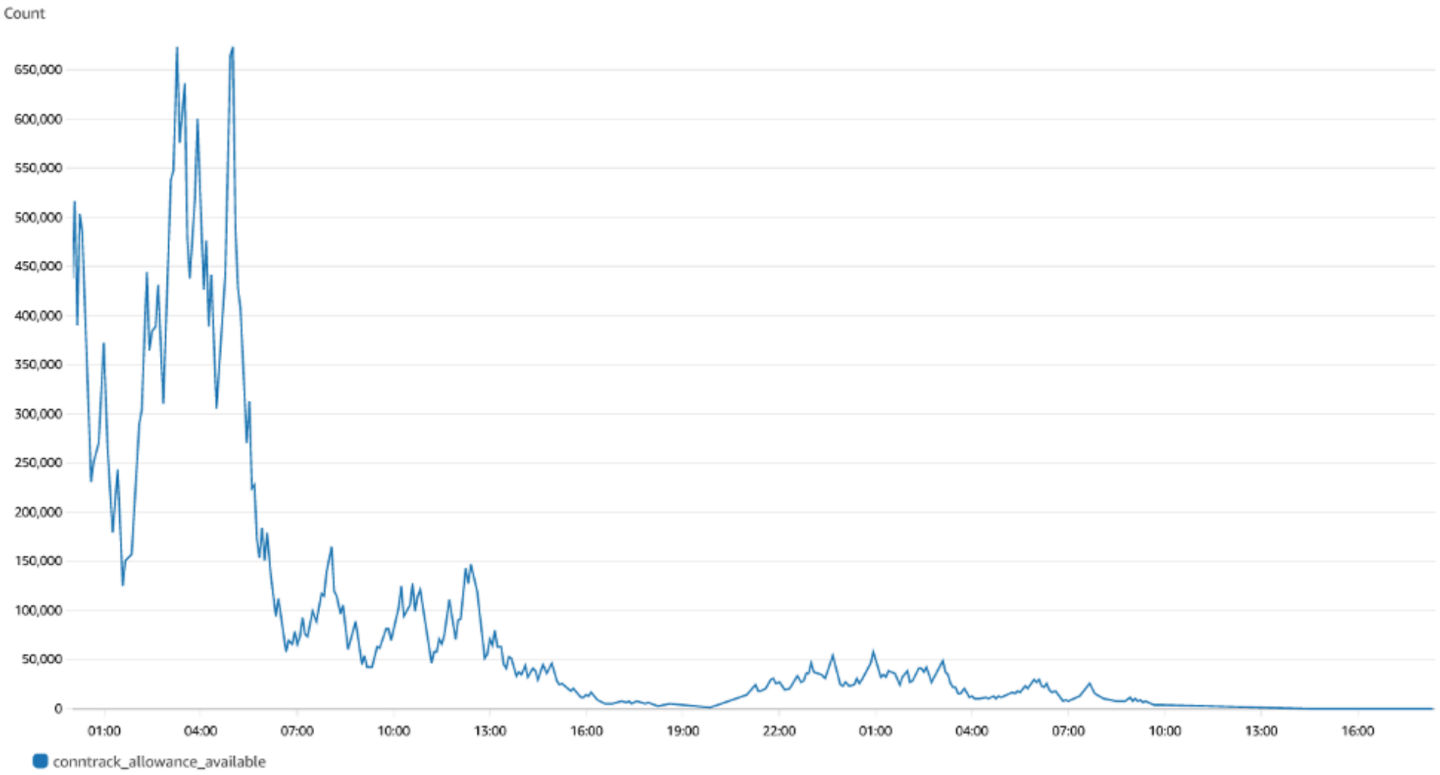



我们可以清楚地看到，由于该值为零，因此没有丢弃任何数据包。

让我们使用查询为 conntrack_allowance_exceed 指标构建一个仪表板。`rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}[30s])`它将生成下面的仪表板。




只要您按此处所述运行 cloudwatch 代理 CloudWatch，conntrack_allowance_exceeded即可在中直观显示该指标。中生成的仪表板 CloudWatch 将如下所示：




我们可以清楚地看到，由于该值为零，因此没有丢弃任何数据包。如果您使用的是 Nitro-based 实例，则可以为 EC2 实例创建类似的控制面板conntrack_allowance_available并主动监控其中的连接。您可以通过在 Amazon Managed Grafana 中配置警报来进一步扩展此范围，以便向 Slack、SNS、Pagerduty 等发送通知。

# 在 IPVS 模式下运行 kube-proxy

IP 虚拟服务器 (IPVS) 模式下的 EKS 解决了运行拥有 1,000 多个服务的大型集群时经常出现的网络延迟问题，kube-proxy 在传统 iptables 模式下运行。此性能问题是按顺序处理每个数据包的 iptables 数据包过滤规则所致。这个延迟问题已在 iptables 的继任者 nftables 中得到解决。但是，截至撰写本文时，kube-proxy 仍在开发中，以利用 nftables。要解决此问题，您可以将集群配置为`kube-proxy`在 IPVS 模式下运行。

## 概述

IPVS 自 Kubernetes 1.11 版本起就已正式发布，它使用哈希表而不是线性搜索来处理数据包，从而为拥有数千个节点和服务的集群提供了效率。IPVS 专为负载平衡而设计，使其成为解决 Kubernetes 网络性能问题的合适解决方案。

IPVS 提供了多种用于将流量分配到后端 pod 的选项。每个选项的详细信息可以在 Kubernetes 官方文档中找到，但下面显示了一个简单的列表。循环和最少连接是 Kubernetes 中最受欢迎的 IPVS 负载平衡选项之一。

```
- rr (Round Robin)
- wrr (Weighted Round Robin)
- lc (Least Connections)
- wlc (Weighted Least Connections)
- lblc (Locality Based Least Connections)
- lblcr (Locality Based Least Connections with Replication)
- sh (Source Hashing)
- dh (Destination Hashing)
- sed (Shortest Expected Delay)
- nq (Never Queue)
```

## 实施

只需几个步骤即可在您的 EKS 集群中启用 IPVS。您需要做的第一件事是确保您的 EKS 工作节点映像已安装 Linux 虚拟服务器管理`ipvsadm`软件包。要在基于 Fedora 的映像（例如 Amazon Linux 2023）上安装此软件包，您可以在工作节点实例上运行以下命令。

```
sudo dnf install -y ipvsadm
```

在基于 Debian 的镜像（例如 Ubuntu）上，安装命令将如下所示。

```
sudo apt-get install ipvsadm
```

接下来，您需要加载上面列出的 IPVS 配置选项的内核模块。我们建议将这些模块写入/etc/modules-load.d/目录内的文件中，这样它们在重启后仍能存活下来。

```
sudo sh -c 'cat << EOF > /etc/modules-load.d/ipvs.conf
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_lc
ip_vs_wlc
ip_vs_lblc
ip_vs_lblcr
ip_vs_sh
ip_vs_dh
ip_vs_sed
ip_vs_nq
nf_conntrack
EOF'
```

您可以运行以下命令将这些模块加载到已经运行的计算机上。

```
sudo modprobe ip_vs
sudo modprobe ip_vs_rr
sudo modprobe ip_vs_wrr
sudo modprobe ip_vs_lc
sudo modprobe ip_vs_wlc
sudo modprobe ip_vs_lblc
sudo modprobe ip_vs_lblcr
sudo modprobe ip_vs_sh
sudo modprobe ip_vs_dh
sudo modprobe ip_vs_sed
sudo modprobe ip_vs_nq
sudo modprobe nf_conntrack
```

> **Note**
>
> 强烈建议将这些工作节点步骤作为工作节点引导过程的一部分，通过用户数据脚本或为构建自定义工作节点 AMI 而执行的任何构建脚本中。

接下来，您将配置集群kube-proxy DaemonSet 以在 IPVS 模式下运行。这是通过将 to ipvs 和 to 设置kube-proxymode为上面列出的负载平衡选项之一ipvs scheduler来完成的，rr例如：对于 Round Robin。

> **Warning**
>
> 这是一种破坏性的更改，应在非工作时间执行。我们建议在初始创建 EKS 集群时进行这些更改，以最大限度地减少影响。

您可以发出 AWS CLI 命令通过更新 kube-proxy E Add-on KS 来启用 IPVS。

```
aws eks update-addon --cluster-name $CLUSTER_NAME --addon-name kube-proxy \
  --configuration-values '{"ipvs": {"scheduler": "rr"}, "mode": "ipvs"}' \
  --resolve-conflicts OVERWRITE
```

或者，您可以通过修改集群kube-proxy-config ConfigMap 中的来实现此目的。

```
kubectl -n kube-system edit cm kube-proxy-config
```

在下面找到该scheduler设置，ipvs并将该值设置为上面列出的 ipvs 负载平衡选项之一，rr例如：对于 Round Robin。找到默认为的mode设置iptables，然后将值更改为ipvs。任一选项的结果都应与以下配置类似。

```
  iptables:
    masqueradeAll: false
    masqueradeBit: 14
    minSyncPeriod: 0s
    syncPeriod: 30s
  ipvs:
    excludeCIDRs: null
    minSyncPeriod: 0s
    scheduler: "rr"
    syncPeriod: 30s
  kind: KubeProxyConfiguration
  metricsBindAddress: 0.0.0.0:10249
  mode: "ipvs"
  nodePortAddresses: null
  oomScoreAdj: -998
  portRange: ""
```

```
  udpIdleTimeout: 250ms
```

如果您的工作节点在进行这些更改之前已加入集群，则需要重新启动 kube DaemonSet-proxy。

```
kubectl -n kube-system rollout restart ds kube-proxy
```

## 验证

您可以通过在其中一个工作节点上发出以下命令来验证您的集群和工作节点是否在 IPVS 模式下运行。

```
sudo ipvsadm -L
```

至少，您应该会看到与以下结果相似的结果，其中显示了 Kubernetes API 服务器服务的条目，网址为 CoreDNS 服务的条目。`10.100.0.1` `10.100.0.10`

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  ip-10-100-0-1.us-east-1. rr
  -> ip-192-168-113-81.us-eas Masq        1      0          0
  -> ip-192-168-162-166.us-ea Masq        1      1          0
TCP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
UDP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
```

> **Note**
>
> 此示例输出来自服务 IP 地址范围为 EKS 集群`10.100.0.0/16`。

# EKS 可扩展性最佳实践

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

本指南为扩展 EKS 集群提供了建议。扩展 EKS 集群的目标是最大限度地提高单个集群可以执行的工作量。与使用多个集群相比，使用单个大型 EKS 集群可以减少运营负载，但在多区域部署、租户隔离和集群升级等方面需要权衡取舍。在本文档中，我们将重点介绍如何使用单个集群实现最大的可扩展性。

## 如何使用本指南

本指南适用于负责在 AWS 中创建和管理 EKS 集群的开发人员和管理员。它侧重于一些通用的 Kubernetes 扩展实践，但它没有具体说明自行管理的 Kubernetes 集群或使用 EKS Anywhere 在 AWS 区域之外运行的集群。

每个主题都有一个简短的概述，然后是大规模运营 EKS 集群的建议和最佳实践。无需按特定顺序阅读主题，未经测试和验证建议在您的集群中是否有效，则不应应用这些建议。

## 了解缩放维度

可扩展性不同于性能和可靠性，在规划集群和工作负载需求时，应将这三者都考虑在内。随着集群的扩展，需要对其进行监控，但本指南不涵盖监控最佳实践。EKS 可以扩展到大规模，但你需要计划如何将集群扩展到 300 个节点或 5,000 个 Pod 以上。这些数字不是绝对数字，但它们来自于与多个用户、工程师和支持专业人员合作使用本指南。

Kubernetes 中的扩展是多维的，没有适用于每种情况的特定设置或建议。我们可以为扩展提供指导的主要领域包括：

E@@ KS 集群中的 Kubernetes 控制平面包括 AWS 为你自动运行和扩展的所有服务（例如 Kubernetes API 服务器）。扩展控制平面是 AWS 的责任，但负责任地使用控制平面是您的责任。

Kubernetes 数据平面扩展处理集群和工作负载所需的 AWS 资源，但它们不在 EKS 控制平面之内。包括 EC2 实例、kubelet 和存储在内的资源都需要随着集群的扩展而进行扩展。

集群服务是在集群内部运行的 Kubernetes 控制器和应用程序，可为您的集群和工作负载提供功能。这些可以是 EKS Add-ons，也可以是您为合规性和集成而安装的其他服务或 Helm 图表。这些服务通常依赖于工作负载，随着工作负载的扩展，您的集群服务将需要随之扩展。

工作负载是您拥有集群的原因，并且应该与集群一起水平扩展。Kubernetes 中工作负载的集成和设置可以帮助集群扩展。Kubernetes 抽象也需要考虑架构问题，例如命名空间和服务。

## 超大缩放比例

如果您要将单个集群扩展到 1,000 个节点或 50,000 个 Pod 以上，我们很乐意与您交谈。我们建议您联系您的支持团队或技术客户经理，与专家取得联系，他们可以帮助您规划和扩展本指南中提供的信息。如果您被选中入门，Amazon EKS 可以在单个集群中支持多达 100,000 个节点。

## Kubernetes 控制平面

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Kubernetes 控制平面由 Kubernetes API 服务器、Kubernetes 控制器管理器、调度器和 Kubernetes 运行所需的其他组件组成。这些组件的可扩展性限制因您在集群中运行的内容而异，但对扩展影响最大的领域包括 Kubernetes 版本、利用率和单个节点扩展。

您可以通过以下两种模式之一运行集群的控制平面，以满足不同的工作负载要求：

- 标准模式-默认情况下，所有 EKS 集群都使用标准模式。控制平面会根据您的工作负载需求自动向上和向下扩展。标准模式可动态分配足够的控制平面容量，是大多数用例的推荐选项。
- 预配置模式 — 如果您的工作负载无法容忍控制平面扩展带来的性能变化，或者如果它们需要非常高的控制平面容量，则可以使用预配置模式。使用预配置模式，您可以预先分配控制平面容量，以随时准备处理苛刻的需求。您可以获得一致且可预测的性能。

在 EKS 预配置模式下，您可以从一组扩展等级（XL、2XL、4XL 和 8XL）中进行选择。在每个层级中，您都可以从集群的控制平面获得高且可预测的性能。预配置模式对于以下用例特别有用：

- Performance-critical 工作负载
- Large-scale AI 和机器学习操作

- 预计需求旺盛的活动
- 需要在暂存和生产之间保持一致性的环境

使用预配置模式，您可以提前预先分配控制平面容量，并受益于以 1 分钟为间隔衡量的增强型 99.99% 服务级别协议 (SLA)。有关 EKS 预配置模式的更多信息，包括等级规格和定价，请参阅 E KS 用户指南中的 EKS 预配置控制平面。

## 限制工作负载和节点爆发

> **Important**
>
> 为了避免在控制平面上达到 API 限制，您应该限制一次将集群大小增加两位数百分比的扩展峰值（例如，1000 个节点增加到 1100 个节点，或者同时增加 4000 到 4500 个 pod）。

EKS 控制平面将随着集群的增长而自动扩展，但是它的扩展速度有限制。首次创建 EKS 集群时，控制平面将无法立即扩展到数百个节点或数千个 Pod。要详细了解 EKS 如何进行扩展改进，请参阅此博客文章。

扩展大型应用程序需要基础设施适应以完全准备就绪（例如，变暖负载均衡器）。要控制扩展速度，请确保根据应用程序的正确指标进行扩展。CPU 和内存缩放可能无法准确预测您的应用程序限制，在 Kubernetes 水平容器自动扩缩器 (HPA) 中使用自定义指标（例如每秒请求数）可能是更好的扩展选项。

要使用自定义指标，请参阅 Kubernetes 文档中的示例。如果您有更高级的扩展需求或需要根据外部来源（例如 AWS SQS 队列）进行扩展，请使用 KEDA 进行基于事件的工作负载扩展。

## 安全地向下扩展节点和 pod

### 替换长时间运行的实例

定期更换节点可以避免配置偏差和只有在延长正常运行时间后才会出现的问题（例如缓慢的内存泄漏），从而保持集群的健康。自动替换将为您提供良好的节点升级和安全补丁流程和实践。如果定期更换集群中的每个节点，那么维护单独的进程以进行持续维护所需的辛勤工作就会减少。

使用 Karpenter 的生存时间 (TTL) 设置在实例运行了指定时间后替换这些实例。自我管理的节点组可以使用该`max-instance-lifetime`设置自动循环节点。托管节点组目前没有此功能，但您可以在此处跟踪请求 GitHub。

### 移除未充分利用的节点

当节点没有正在运行的工作负载时，你可以使用 Kubernetes 集群自动扩缩器中的缩减阈值将其移除，`--scale-down-utilization-threshold`或者在 Karpenter 中使用配置器设置。`ttlSecondsAfterEmpty`

### 使用 pod 中断预算和安全关闭节点

从 Kubernetes 集群中移除 Pod 和节点需要控制器对多个资源进行更新（例如）。 EndpointSlices当更改传播到控制器时，频繁或过快地执行此操作可能会导致 API 服务器限制和应用程序中断。当集群中的节点被移除或重新调度时，Pod 中断预算是减慢流失速度以保护工作负载可用性的最佳实践。

## 运行 Kubectl 时使用 Client-Side 缓存

如果使用 kubectl 命令效率低下，可能会给 Kubernetes API 服务器增加额外的负载。你应该避免运行重复使用 kubectl 的脚本或自动化（例如在 for 循环中），或者在没有本地缓存的情况下运行命令。

`kubectl`具有客户端缓存，用于缓存来自集群的发现信息，以减少所需的 API 调用量。默认情况下，缓存处于启用状态，并且每 10 分钟刷新一次。

如果您从容器或没有客户端缓存的情况下运行 kubectl，则可能会遇到 API 限制问题。建议通过挂载来保留您的集群缓存，`--cache-dir`以避免进行不必要的 API 调用。

## 禁用 kubectl 压缩

在你的 kubeconfig 文件中禁用 kubectl 压缩可以降低 API 和客户端 CPU 的使用率。默认情况下，服务器将压缩发送到客户端的数据以优化网络带宽。这会增加每个请求在客户端和服务器上的 CPU 负载，如果您有足够的带宽，禁用压缩可以减少开销和延迟。要禁用压缩，你可以使用该`--disable-compression=true`标志或在 kubeconfig 文件`disable-compression: true`中设置该标志。

```
apiVersion: v1
clusters:
- cluster:
    server: serverURL
    disable-compression: true
  name: cluster
```

# 分片集群自动扩缩器

Kubernetes 集群自动扩缩器已经过测试，可以扩展到 1000 个节点。在节点超过 1000 个的大型集群上，建议在分片模式下运行集群自动扩缩程序的多个实例。每个 Cluster Autoscaler 实例都配置为扩展一组节点组。以下示例显示了为每个 4 级节点组配置的 2 个集群自动缩放配置。

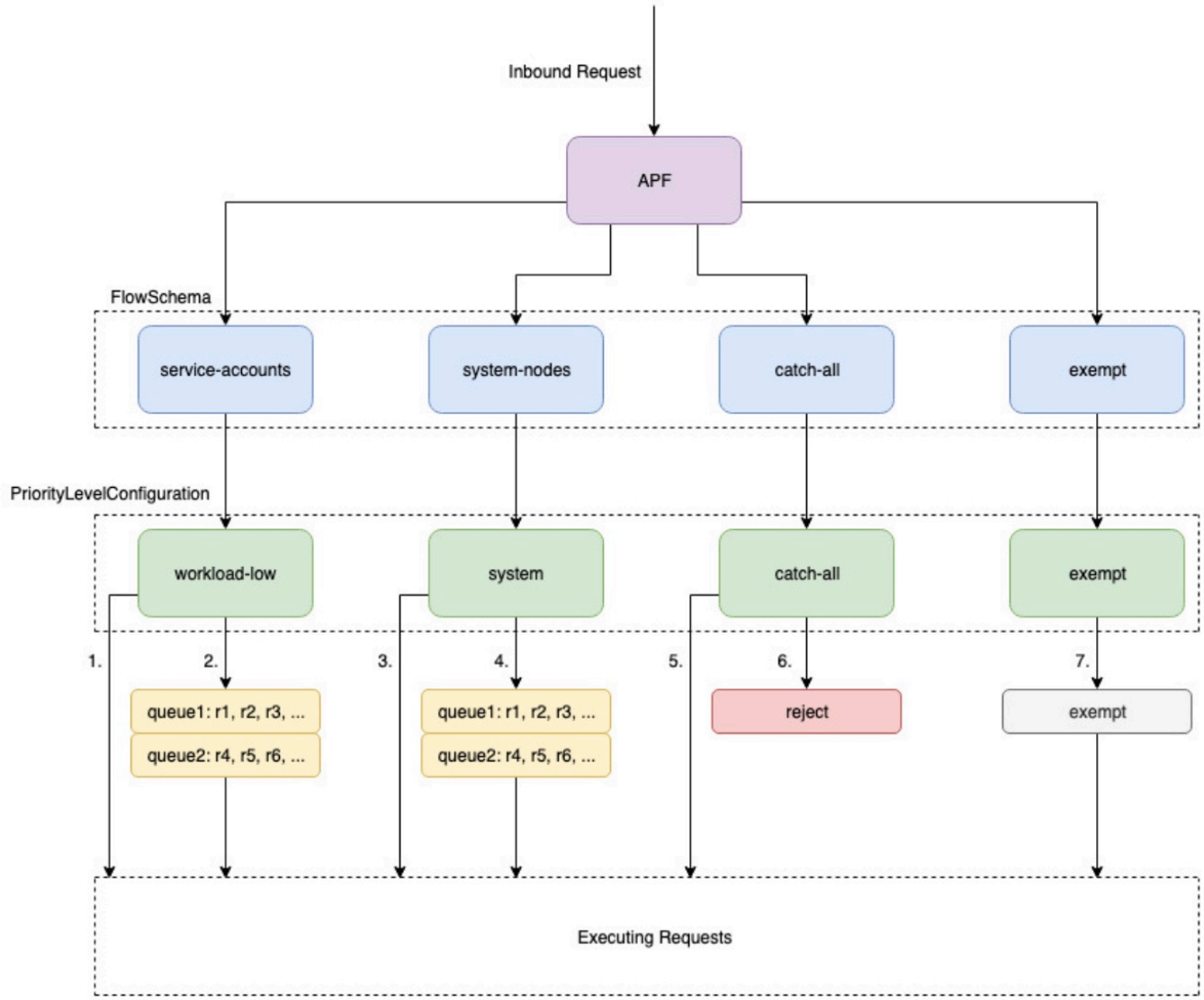
ClusterAutoscaler-1

```
autoscalingGroups:
- name: eks-core-node-grp-20220823190924690000000011-80c1660e-030d-476d-cb0d-
d04d585a8fcb
  maxSize: 50
  minSize: 2
- name: eks-data_m1-20220824130553925600000011-5ec167fa-ca93-8ca4-53a5-003e1ed8d306
  maxSize: 450
  minSize: 2
- name: eks-data_m2-20220824130733258600000015-aac167fb-8bf7-429d-d032-e195af4e25f5
  maxSize: 450
  minSize: 2
- name: eks-data_m3-20220824130553914900000003-18c167fa-ca7f-23c9-0fea-f9edefbda002
  maxSize: 450
  minSize: 2
```

ClusterAutoscaler-2

```
autoscalingGroups:
- name: eks-data_m4-20220824130553925500000000f-5ec167fa-ca86-6b83-ae9d-1e07ade3e7c4
  maxSize: 450
  minSize: 2
- name: eks-data_m5-20220824130744542100000017-02c167fb-a1f7-3d9e-a583-43b4975c050c
  maxSize: 450
  minSize: 2
- name: eks-data_m6-20220824130553924300000000d-9cc167fa-ca94-132a-04ad-e43166cef41f
  maxSize: 450
  minSize: 2
- name: eks-data_m7-20220824130553921000000009-96c167fa-ca91-d767-0427-91c879ddf5af
  maxSize: 450
  minSize: 2
```

# API 优先级和公平性




1. If workload-low has not reached its concurrency limit, the request will be dispatched immediately.
2. If workload-low has reached its concurrency limit, the request will be queued prior to dispatch.
3. If system has not reached its concurrency limit, the request will be dispatched immediately.
4. If system has reached its concurrency limit, the request will be queued prior to dispatch.
5. If catch-all has not reached its concurrency limit, the request will be dispatched immediately.
6. If catch-all has reached its concurrency limit, the request will be dropped.
7. Requests mapping to exempt will always be dispatched.

## 概述

为了防止自己在请求增加期间过载，API 服务器限制了在给定时间内可能未完成的飞行中请求的数量。超过此限制后，API 服务器将开始拒绝请求，并将 “请求过多” 的 429 HTTP 响应代码返回给客户端。

服务器丢弃请求并让客户端稍后再试比服务器端对请求数量没有限制和控制平面过载更可取，后者可能会导致性能下降或不可用。

Kubernetes 用来配置如何在不同请求类型之间划分这些机上请求的机制称为 API 优先级和公平性。API 服务器通过将和标志指定的值相加来配置它可以接受的机上请求总数。`--max-requests-inflight --max-mutating-requests-inflight`EKS 对这些标志使用默认值 400 和 200 个请求，允许在给定时间总共发送 600 个请求。但是，由于它会将控制平面扩展到更大的大小以应对利用率的增加和工作负载流失，因此它相应地将机上请求配额一直增加到2000年（可能会发生变化）。APF 规定了如何在不同的请求类型之间进一步细分这些机上请求配额。请注意，EKS 控制平面高度可用，每个集群至少注册了 2 个 API 服务器。这意味着您的集群可以处理的飞行中请求总数是每个 kube-apiserver 设置的机上配额的两倍（如果进一步横向扩展，则更高）。在最大的EKS集群 requests/second 上，这相当于数千个。

两种名为 “ PriorityLevelConfigurations 和 FlowSchemas” 的 Kubernetes 对象配置如何在不同的请求类型之间分配请求总数。这些对象由 API 服务器自动维护，EKS 使用给定的 Kubernetes 次要版本中这些对象的默认配置。 PriorityLevelConfigurations 仅占允许的请求总数的一小部分。例如，在总共 600 个请求中 PriorityLevelConfiguration ，分配了工作负载最高的 98 个。分配给所有请求的总和 PriorityLevelConfigurations 将等于 600（或略高于 600，因为如果向给定级别授予请求的一小部分，API 服务器将向上舍入）。要检查集群 PriorityLevelConfigurations 中的以及分配给每个集群的请求数，可以运行以下命令。以下是 EKS 1.32 的默认设置：

```
$ kubectl get --raw /metrics | grep apiserver_flowcontrol_nominal_limit_seats
apiserver_flowcontrol_nominal_limit_seats{priority_level="catch-all"} 13
apiserver_flowcontrol_nominal_limit_seats{priority_level="exempt"} 0
apiserver_flowcontrol_nominal_limit_seats{priority_level="global-default"} 49
apiserver_flowcontrol_nominal_limit_seats{priority_level="leader-election"} 25
apiserver_flowcontrol_nominal_limit_seats{priority_level="node-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="system"} 74
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245
```

第二种物体是 FlowSchemas。具有给定属性集的 API 服务器请求归入同一组属性 FlowSchema。这些属性包括经过身份验证的用户或请求的属性，例如 API 组、命名空间或资源。A FlowSchema 还指定应映射到哪 PriorityLevelConfiguration 种类型的请求。这两个对象一起说：“我希望这种类型的请求计入机上请求的份额。” 当请求到达 API 服务器时，它将检查每个请求， FlowSchemas 直到找到一个与所有必需属性相匹配的请求。如果多个请求 FlowSchemas 匹配，API 服务器将选择匹配优先级最小的，该优先级被指定为对象中的一个属性。 FlowSchema

PriorityLevelConfigurations 可以使用以下命令查看 FlowSchemas 到的映射：

```
$ kubectl get flowschemas
NAME                           PRIORITYLEVEL     MATCHINGPRECEDENCE
 DISTINGUISHERMETHOD   AGE     MISSINGPL
exempt                         exempt            1                  <none>
     7h19m   False
eks-exempt                     exempt            2                  <none>
     7h19m   False
probes                         exempt            2                  <none>
     7h19m   False
system-leader-election         leader-election   100                ByUser
     7h19m   False
endpoint-controller            workload-high     150                ByUser
     7h19m   False
workload-leader-election       leader-election   200                ByUser
     7h19m   False
system-node-high               node-high         400                ByUser
     7h19m   False
system-nodes                   system            500                ByUser
     7h19m   False
kube-controller-manager        workload-high     800                ByNamespace
     7h19m   False
kube-scheduler                 workload-high     800                ByNamespace
     7h19m   False
kube-system-service-accounts   workload-high     900                ByNamespace
     7h19m   False
eks-workload-high              workload-high     1000               ByUser
     7h14m   False
service-accounts               workload-low      9000               ByUser
     7h19m   False
global-default                 global-default    9900               ByUser
     7h19m   False
catch-all                      catch-all         10000              ByUser
     7h19m   False
```

PriorityLevelConfigurations 可以有 “队列”、“拒绝” 或 “豁免” 的类型。对于 Queue 和 Reject 类型，将对该优先级的最大机上请求数施加限制，但是，当达到该限制时，行为会有所不同。例如，workload-high PriorityLevelConfiguration 使用 Queue 类型，有 98 个请求可供控制器管理器、端点控制器、调度器、eks 相关控制器以及在 kube-system 命名空间中运行的 pod 使用。由于使用了 Queue 类型，API 服务器将尝试将请求保留在内存中，并希望在这些请求超时之前将正在运行的请求数量降至 98 以下。如果给定请求在队列中超时，或者已经排队的请求太多，API 服务器别无选择，只能放弃该请求并向客户端返回 429。请注意，排队可能会阻止请求接收 429，但需要权衡请求的端到端延迟增加。

现在考虑一下映射到包罗万 FlowSchema 象的 “拒绝” 类型 PriorityLevelConfiguration 。如果客户端达到 13 个机上请求的限制，API 服务器将不会进行排队，而是使用 429 响应代码立即丢弃请求。最后，映射到类型为 Exp PriorityLevelConfiguration ection 的请求将永远不会收到 429，并且始终会立即分发。这用于高优先级请求，例如 healthz 请求或来自 system: masters 组的请求。

## 监控 APF 和已删除的请求

要确认是否有任何请求因 APF 而被丢弃，`apiserver_flowcontrol_rejected_requests_total`可以监控 API 服务器指标以检查受影响的 FlowSchemas 和 PriorityLevelConfigurations。例如，此指标显示，由于请求在工作负载较低的队列中超时，来自服务帐号的 100 个请求 FlowSchema 被丢弃：

```
% kubectl get --raw /metrics | grep apiserver_flowcontrol_rejected_requests_total
apiserver_flowcontrol_rejected_requests_total{flow_schema="service-
accounts",priority_level="workload-low",reason="time-out"} 100
```

要检查给定对象与接收 429 的 PriorityLevelConfiguration 距离有多近，或者由于排队而出现延迟增加，您可以比较并发限制和正在使用的并发数之间的差异。在此示例中，我们有一个包含 100 个请求的缓冲区。

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_nominal_limit_seats.*workload-low'
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245

% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_executing_seats.*workload-low'
apiserver_flowcontrol_current_executing_seats{flow_schema="service-
accounts",priority_level="workload-low"} 145
```

要检查给定对象 PriorityLevelConfiguration 是否遇到排队但请求不一定被丢弃的情况，`apiserver_flowcontrol_current_inqueue_requests`可以参考以下指标：

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_inqueue_requests.*workload-low'
apiserver_flowcontrol_current_inqueue_requests{flow_schema="service-
accounts",priority_level="workload-low"} 10
```

其他有用的 Prometheus 指标包括：

- apiserver_flowcontrol_dispatched_request_t

- apiserver_flowcontrol_requestion_exection
- apiserver_flowcontrol_request_wait_duration_se

有关 APF 指标的完整列表，请参阅上游文档。

## 防止请求被丢弃

通过更改工作负载来防止 429 秒

当 APF 因给定请求 PriorityLevelConfiguration 超出其允许的最大机上请求数而丢弃请求时，受影响的客户端 FlowSchemas 可以减少在给定时间执行的请求数。这可以通过减少在发生 429 的时段内发出的请求总数来实现。请注意，长时间运行的请求（例如昂贵的列表调用）尤其成问题，因为它们在整个执行期间都算作飞行中请求。减少这些昂贵的请求的数量或优化这些列表调用的延迟（例如，通过减少每个请求获取的对象数量或切换到使用监视请求）可以帮助减少给定工作负载所需的总并发量。

通过更改 APF 设置来防止 429 秒

> **Warning**
>
> 只有在知道自己在做什么时才更改默认 APF 设置。APF 设置配置不当可能会导致 API 服务器请求丢失和工作负载严重中断。

防止请求被丢弃的另一种方法是更改默认值 FlowSchemas 或在 EKS 集群上 PriorityLevelConfigurations 安装的设置。EKS 为 FlowSchemas 给定的 Kubernetes 次要版本安装上游默认设置。 PriorityLevelConfigurations 如果修改了这些对象，API 服务器会自动将这些对象协调回其默认值，除非这些对象的以下注释设置为 false：

```
metadata:
  annotations:
    apf.kubernetes.io/autoupdate-spec: "false"
```

简而言之，可以将 APF 设置修改为：

- 为您关心的请求分配更多机上容量。
- 隔离不必要或昂贵的请求，这些请求可能会导致其他请求类型的容量不足。

这可以通过更改默认值 FlowSchemas 和 PriorityLevelConfigurations 或通过创建这些类型的新对象来实现。运营商可以增加相关 PriorityLevelConfigurations 对象ConcurrencyShares 的保证值，以增加分

配给它们的机上请求的比例。此外，如果应用程序能够处理因请求在调度之前排队而造成的额外延迟，则在给定时间可以排队的请求数量也可以增加。

或者，可以创建特定于客户工作负载的新 PriorityLevelConfigurations 对象 FlowSchema 和对象。请注意，ConcurrencyShares 向现有 PriorityLevelConfigurations 或新存储桶分配更有保障的请求 PriorityLevelConfigurations 将导致其他存储桶可以处理的请求数量减少，因为每个 API 服务器的总体限制将保持在 600 个正在进行中。

更改 APF 默认值时，应在非生产集群上监控这些指标，以确保更改设置不会导致意外的 429：

1. `apiserver_flowcontrol_rejected_requests_total`应监控所有人的指标， FlowSchemas 以确保没有存储桶开始丢弃请求。
2. `apiserver_flowcontrol_current_executing_seats`应比较`apiserver_flowcontrol_nominal_limit_seats`和的值，以确保使用的并发不会因为突破该优先级的限制而面临风险。

定义新的 and 的一个常见用例 PriorityLevelConfiguration 是 FlowSchema 隔离。假设我们想将来自 pod 的长时间运行的列表事件调用隔离到它们自己的请求份额。这将防止来自使用现有服务帐户 FlowSchema 的 pod 的重要请求接收 429 并导致请求容量不足。回想一下，飞行中请求的总数是有限的，但是，此示例显示可以修改 APF 设置以更好地划分给定工作负载的请求容量：

用于隔离列表事件请求的示例 FlowSchema 对象：

```
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchema
metadata:
  name: list-events-default-service-accounts
spec:
  distinguisherMethod:
    type: ByUser
  matchingPrecedence: 8000
  priorityLevelConfiguration:
    name: catch-all
  rules:
  - resourceRules:
    - apiGroups:
      - '*'
      namespaces:
      - default
      resources:
```

```
    - events
    verbs:
    - list
  subjects:
  - kind: ServiceAccount
    serviceAccount:
      name: default
      namespace: default
```

- 这将 FlowSchema 捕获默认命名空间中服务帐号发出的所有列表事件调用。
- 匹配的优先级 8000 低于现有服务帐户使用的值 9000， FlowSchema 因此这些列表事件调用将匹配列表事件-默认服务帐户而不是服务帐户。
- 我们正在使用包罗万象 PriorityLevelConfiguration 来隔离这些请求。此存储桶仅允许这些长时间运行的列表事件调用使用 13 个飞行中请求。当 Pod 尝试同时发出超过 13 个这样的请求时，它们就会开始收到 429。

## 在 API 服务器中检索资源

对于任何规模的集群，从 API 服务器获取信息都是一种预期行为。在扩展集群中的资源数量时，请求频率和数据量很快就会成为控制平面的瓶颈，并导致 API 延迟和缓慢。根据延迟的严重程度，如果不小心，它会导致意外停机。

了解您的要求以及避免此类问题的第一步是多久一次。以下是根据扩展最佳实践限制查询量的指南。本节中的建议是按顺序提供的，从已知可以最佳扩展的选项开始。

### 使用共享举报人

在构建与 Kubernetes API 集成的控制器和自动化时，您通常需要从 Kubernetes 资源中获取信息。如果您定期轮询这些资源，可能会导致 API 服务器负载过重。

使用 client-go 库中的告密器可以让你根据事件监视资源的变化，而不是轮询更改。告密者通过对事件和更改使用共享缓存来进一步减少负载，因此监视相同资源的多个控制器不会增加额外的负载。

控制器应避免在没有标签和字段选择器的情况下轮询集群范围内的资源，尤其是在大型集群中。每个未过滤的民意调查都需要通过 API 服务器从 etcd 发送大量不必要的数据，然后由客户端进行过滤。通过根据标签和命名空间进行筛选，可以减少 API 服务器为完成发送到客户端的请求和数据而需要执行的工作量。

## 优化 Kubernetes API 的使用情况

使用自定义控制器或自动化调用 Kubernetes API 时，请务必将调用限制在所需的资源上。没有限制，您可能会在 API 服务器和 etcd 上造成不必要的负载。

建议尽可能使用 watch 参数。如果没有参数，则默认行为是列出对象。要使用监视而不是列表，您可以在 API 请求`?watch=true`的末尾追加。例如，要使用手表获取默认命名空间中的所有 pod，请使用：

```
/api/v1/namespaces/default/pods?watch=true
```

如果您要列出对象，则应限制所列内容的范围和返回的数据量。您可以通过在请求中添加`limit=500`参数来限制返回的数据。`fieldSelector`参数和`/namespace/`路径对于确保列表的范围根据需要缩小范围很有用。例如，要仅列出默认命名空间中正在运行的 Pod，请使用以下 API 路径和参数。

```
/api/v1/namespaces/default/pods?fieldSelector=status.phase=Running&limit=500
```

或者列出所有正在使用以下命令运行的 pod：

```
/api/v1/pods?fieldSelector=status.phase=Running&limit=500
```

限制监视调用或列出的对象的另一种选择是使用它 `resourceVersions`，你可以在 Kubernetes 文档中阅读相关内容。如果不带任何`resourceVersion`参数，您将收到最新的可用版本，该版本需要 etcd 法定读取，这是数据库中最昂贵和最慢的读取。ResourceVersion 取决于你要查询的资源，这些资源可以在现场找到。`metadata.resourseVersion`如果使用监视通话，而不仅仅是列出通话，也建议这样做

有一种特殊功能`resourceVersion=0`可以从 API 服务器缓存中返回结果。这可以减少 etcd 的负载，但它不支持分页。

```
/api/v1/namespaces/default/pods?resourceVersion=0
```

建议使用 watch，将 ResourceVersion 设置为从其先前的列表或监视中收到的最新已知值。这是在 client-go 中自动处理的。但是，如果您使用的是其他语言的 k8s 客户端，建议您仔细检查一下。

```
/api/v1/namespaces/default/pods?watch=true&resourceVersion=362812295
```

如果你在没有任何参数的情况下调用 API，它将是 API 服务器和 etcd 资源最密集的。此调用将在不进行分页或限制范围的情况下获取所有命名空间中的所有 pod，并且需要从 etcd 读取法定人数。

```
/api/v1/pods
```

## 防止 DaemonSet 雷鸣般的牛群

A DaemonSet 可确保所有（或部分）节点都运行 Pod 的副本。当节点加入集群时，守护程序集控制器会为这些节点创建 Pod。当节点离开集群时，这些 Pod 会被垃圾回收。删除 a DaemonSet 会清理它创建的 pod。

a 的一些典型用法 DaemonSet 是：

- 在每个节点上运行集群存储守护程序
- 在每个节点上运行日志收集守护程序
- 在每个节点上运行节点监视守护程序

在拥有数千个节点的集群上 DaemonSet，创建新节点 DaemonSet、更新节点或增加节点数量可能会导致控制平面上的负载过高。如果 DaemonSet Pod 在 Pod 启动时发出昂贵的 API 服务器请求，则大量并发请求可能会导致控制层面的资源使用率过高。

在正常操作中，您可以使用`RollingUpdate`来确保逐步推出新 DaemonSet pod。使用`RollingUpdate`更新策略，在你更新 DaemonSet 模板后，控制器会以受控的方式杀死旧 DaemonSet 的 p DaemonSet od 并自动创建新的 pod。在整个更新过程中 DaemonSet ，每个节点上最多只能运行一个 pod。您可以通过将设置`maxUnavailable`为 1、0 和 `minReadySeconds` 60 `maxSurge` 来执行逐步部署。如果您未指定更新策略，Kubernetes 将默认为创建一个，其值`RollingUpdate``maxUnavailable`为 1、`maxSurge`为 0 和 `minReadySeconds` 0。

```
minReadySeconds: 60
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 0
    maxUnavailable: 1
```

如果已经创建了新的 DaemonSet pod，并且所有节点上 DaemonSet 的 pod 数量都达到预期，则`RollingUpdate`可以确保逐步推出新`Ready`的 pod。在战略未涵盖的某些条件下，可能会出现雷鸣般的牛群问题。`RollingUpdate`

防止在创建时出现雷鸣般的牛群 DaemonSet

默认情况下，无论RollingUpdate配置如何，kube-controller-manager 中的 daemonset-controller 在创建新节点时都会同时为所有匹配的节点创建 pod。 DaemonSet要在创建 pod 后强制逐步推出 pod DaemonSet，你可以使用NodeSelector或NodeAffinity。这将创建一个匹配零个节点的 DaemonSet ，然后你可以逐渐更新节点，使它们有资格以受控的速率从中 DaemonSet 运行 Pod。你可以遵循这种方法：

- 为的所有节点添加标签run-daemonset=false。

```
kubectl label nodes --all run-daemonset=false
```

- 使用 DaemonSet 与任何没有run-daemonset=false标签的节点相匹配的NodeAffinity设置来创建。最初，这将导致你 DaemonSet 没有相应的 pod。

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: NotIn
          values:
          - "false"
```

- 以受控的速率从节点上移除run-daemonset=false标签。你可以用这个 bash 脚本作为示例：

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes" | jq -r '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Removing run-daemonset label from node $node"
   kubectl label nodes $node run-daemonset-
   sleep 5
done
```

- （可选）从 DaemonSet 对象中移除该NodeAffinity设置。请注意，由于 DaemonSet 模板已更改，这也会触发RollingUpdate并逐渐替换所有现有的 DaemonSet pod。

防止节点横向扩展时出现雷鸣般的牛群

与 DaemonSet 创建类似，快速创建新节点可能会导致大量 DaemonSet Pod 同时启动。您应该以受控的速率创建新节点，以便控制器以相同的速率创建 DaemonSet Pod。如果无法做到这一点，则可以使用使新节点最初不符合现有 DaemonSet 节点的资格。NodeAffinity接下来，你可以逐渐为新节点添加标签，这样 daemonset-controller 就可以以受控的速率创建 Pod。你可以遵循这种方法：

- 为所有现有节点添加标签 run-daemonset=true

```
kubectl label nodes --all run-daemonset=true
```

- DaemonSet 使用设置更新您的NodeAffinity设置，以匹配任何带有run-daemonset=true标签的节点。请注意，由于 DaemonSet 模板已更改，这也会触发RollingUpdate并逐渐替换所有现有的 DaemonSet pod。您应该等RollingUpdate到完成后再进入下一步。

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: In
          values:
          - "true"
```

- 在集群中创建新节点。请注意，这些节点没有run-daemonset=true标签，因此 DaemonSet 不会匹配这些节点。
- 以受控的速率将run-daemonset=true标签添加到您的新节点（当前没有run-daemonset标签）。你可以用这个 bash 脚本作为示例：

```
#!/bin/bash
```

```
nodes=$(kubectl get --raw "/api/v1/nodes?labelSelector=%21run-daemonset" | jq -r
 '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Adding run-daemonset=true label to node $node"
   kubectl label nodes $node run-daemonset=true
   sleep 5
done
```

- 或者，从 DaemonSet 对象中移除NodeAffinity设置并从所有节点上移除run-daemonset标签。

防止更新时出现雷鸣般的牛群 DaemonSet

RollingUpdate策略只会尊重符合条件的 DaemonSet pod 的maxUnavailable设置Ready。如果 a 只 DaemonSet 有 NotReady pod 或很大比例的 NotReady pod，并且您更新了其模板，则守护程序控制器将同时为任何 pod 创建新的 pod。NotReady如果有大量的 NotReady pod，这可能会导致雷鸣般的牛群问题，例如，如果 pod 持续崩溃循环或无法提取图像。

要在更新 a DaemonSet 且有 pod 时强制逐步推出 NotReady pod，可以暂时将更新策略 DaemonSet 从 from 更改RollingUpdate为OnDelete。在更新 DaemonSet 模板后OnDelete，控制器会在您手动删除旧的 pod 之后创建新的 pod，这样您就可以控制新 pod 的推出了。你可以遵循这种方法：

- 检查您的NotReady吊舱里是否有吊舱 DaemonSet。
- 如果不是，您可以放心地更新 DaemonSet 模板，该RollingUpdate策略将确保逐步推出。
- 如果是，则应先更新您的策略 DaemonSet 以使用该OnDelete策略。

```
updateStrategy:
  type: OnDelete
```

- 接下来，使用所需的更改更新您的 DaemonSet 模板。
- 更新后，您可以通过以受控的速率发出删除 pod 请求来删除旧 DaemonSet pod。你可以用这个 bash 脚本作为示例，其中 kube-system 命名空间中的 DaemonSet 名字是 fluentd-elasticsearch：

```
#!/bin/bash

daemonset_pods=$(kubectl get --raw "/api/v1/namespaces/kube-system/pods?
labelSelector=name%3Dfluentd-elasticsearch" | jq -r '.items | .[].metadata.name')
```

```
for pod in ${daemonset_pods[@]}; do
   echo "Deleting pod $pod"
   kubectl delete pod $pod -n kube-system
   sleep 5
done
```

- 最后，你可以更新 DaemonSet 到之前的`RollingUpdate`策略。

# Kubernetes 数据平面

选择 EC2 实例类型可能是客户面临的最艰难的决定之一，因为在具有多个工作负载的集群中。没有放之四海而皆准的解决方案。以下是一些技巧，可帮助您避免在扩展计算时遇到的常见陷阱。

## 自动节点自动缩放

我们建议您使用节点自动缩放，这样可以减少工作量并与 Kubernetes 深度集成。对于大型@@ 集群，建议使用托管节点组和 Karpenter。

托管节点组将为您提供 Amazon EC2 Auto Scaling 组的灵活性，并在托管升级和配置方面带来更多好处。它可以使用 Kubernetes 集群自动扩缩器进行扩展，对于有各种计算需求的集群来说，这是一种常见的选择。

Karpenter 是一款由 AWS 创建的开源、工作负载原生节点自动扩缩器。它根据资源（例如 GPU）的工作负载要求以及污点和容忍度（例如区域分布）来扩展集群中的节点，而无需管理节点组。节点直接从 EC2 创建，这避免了默认的节点组配额（每组 450 个节点），并提供了更大的实例选择灵活性，同时减少了运营开销。我们建议客户尽可能使用 Karpenter。

## 使用许多不同的 EC2 实例类型

每个 AWS 区域每种实例类型的可用实例数量有限。如果您创建的集群仅使用一种实例类型，并且将节点数量扩展到该区域的容量之外，则会收到一条错误消息，提示没有可用的实例。为避免此问题，您不应任意限制可在集群中使用的实例类型。

默认情况下，Karpenter 将使用一系列兼容的实例类型，并将在配置时根据待处理的工作负载要求、可用性和成本选择实例。您可以扩大`karpenter.k8s.aws/instance-category`密钥中使用的实例类型的列表NodePools。

Kubernetes 集群自动扩缩器要求节点组的大小必须相似，这样它们才能持续扩展。您应该根据 CPU 和内存大小创建多个组，并分别对其进行扩展。使用 ec2-instance-selector 来识别您的节点组大小相似的实例。

```
ec2-instance-selector --service eks --vcpus-min 8 --memory-min 16
a1.2xlarge
a1.4xlarge
a1.metal
c4.4xlarge
c4.8xlarge
c5.12xlarge
c5.18xlarge
c5.24xlarge
c5.2xlarge
c5.4xlarge
c5.9xlarge
c5.metal
```

## 最好使用更大的节点来减少 API 服务器的负载

在决定使用哪些实例类型时，较少的大型节点会减少 Kubernetes 控制平面的负载，因为运行的 kubelet 会更少。 DaemonSets 但是，大型节点可能无法像较小的节点那样得到充分利用。应根据您的工作负载可用性和规模要求来评估节点大小。

一个包含三个 u-24tb1.metal 实例（24 TB 内存和 448 个内核）的集群有 3 个 kubelet，默认情况下每个节点的容量限制为 110 个。如果你的 pod 每个 pod 使用 4 个内核，那么这可能是预料之中的（4 个内核 x 110 = 440 cores/node）。对于 3 节点集群，您处理实例事件的能力会很低，因为 1 个实例中断可能会影响 1/3 集群。你应该在工作负载中指定节点要求和 Pod 分布，这样 Kubernetes 调度器就可以正确放置工作负载。

工作负载应通过污点、容忍度和，来定义他们需要的资源和所需的可用性。PodTopologySpread他们应该选择可以充分利用并满足可用性目标的最大节点，以减少控制平面负载、降低操作和降低成本。

如果资源可用，Kubernetes 调度器将自动尝试将工作负载分散到可用区和主机上。如果没有可用容量，Kubernetes 集群自动扩缩程序将尝试在每个可用区中均匀添加节点。除非工作负载指定了其他要求，否则 Karpenter 将尝试以尽可能快和低廉的价格添加节点。

要强制工作负载随调度器分散并跨可用区创建新节点，应使用拓扑SpreadConstraints：

```
spec:
  topologySpreadConstraints:
```

```
    - maxSkew: 3
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
    - maxSkew: 2
      topologyKey: "kubernetes.io/hostname"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
```

## 使用相似的节点大小以获得一致的工作负载性能

工作负载应定义它们需要在多大小的节点上运行，以实现稳定的性能和可预测的扩展。请求 500 米 CPU 的工作负载在具有 4 个内核的实例和具有 16 个内核的实例上的性能会有所不同。避免使用像 T 系列实例这样的可突发性 CPU 的实例类型。

为了确保您的工作负载获得稳定的性能，工作负载可以使用支持的 Karpenter 标签来定位特定的实例大小。

```
kind: deployment
...
spec:
  template:
    spec:
    containers:
    nodeSelector:
      karpenter.k8s.aws/instance-size: 8xlarge
```

使用 Kubernetes 集群自动扩缩器在集群中调度的工作负载应根据标签匹配将节点选择器与节点组进行匹配。

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: eks.amazonaws.com/nodegroup
            operator: In
```

```
            values:
            - 8-core-node-group    # match your node group name
```

## 高效使用计算资源

计算资源包括 EC2 实例和可用区。有效使用计算资源将提高您的可扩展性、可用性和性能，并降低总成本。在具有多个应用程序的自动缩放环境中，很难预测资源的有效使用情况。Karpenter 旨在根据工作负载需求按需配置实例，从而最大限度地提高利用率和灵活性。

Karpenter 允许工作负载声明其所需的计算资源类型，而无需先创建节点组或为特定节点配置标签污点。有关更多信息，请参阅 Karpenter 最佳实践。考虑在 Karpenter 配置器中启用整合，以替换未充分利用的节点。

## 自动更新亚马逊系统映像 (AMI)

使工作节点组件保持最新状态将确保您拥有最新的安全补丁和与 Kubernetes API 兼容的功能。更新 kubelet 是 Kubernetes 功能最重要的组件，但是自动化操作系统、内核和本地安装的应用程序补丁将减少在扩展时的维护工作。

建议你为节点映像使用最新的亚马逊 EKS 优化的亚马逊 Linux 2 或亚马逊 EKS 优化的 Bottlerocket AM I。Karpenter 将自动使用最新的可用的 AMI 在集群中配置新节点。托管节点组将在节点组更新期间更新 AMI，但不会在节点配置时更新 AMI ID。

对于托管节点组，当补丁版本可用时，您需要使用新的 AMI ID 更新 Auto Scaling 组 (ASG) 启动模板。AMI 次要版本（例如 1.23.5 到 1.24.3）将作为节点组的升级在 EKS 控制台和 API 中提供。补丁发布版本（例如 1.23.5 到 1.23.6）不会作为节点组的升级提供。如果您想让您的节点组与 AMI 补丁版本保持同步，则需要创建新的启动模板版本，并让节点组用新的 AMI 版本替换实例。

您可以从此页面找到最新的可用的 AMI，也可以使用 AWS CLI。

```
aws ssm get-parameter \
  --name /aws/service/eks/optimized-ami/1.24/amazon-linux-2/recommended/image_id \
  --query "Parameter.Value" \
  --output text
```

## 为容器使用多个 EBS 卷

根据卷的类型 input/output （例如 gp3I/O）和磁盘的大小，EBS 卷具有 () 配额。如果您的应用程序与主机共享一个 EBS 根卷，则可能会耗尽整个主机的磁盘配额，并导致其他应用程序等待可用容量。如

果应用程序将文件写入其覆盖分区，从主机装载本地卷，以及登录到标准输出 (STDOUT)，则会写入磁盘，具体取决于所使用的日志代理。

为避免磁盘 I/O 耗尽，应将第二个卷挂载到容器状态文件夹（例如/run/containerd），使用单独的 EBS 卷存储工作负载，并禁用不必要的本地日志记录。

要使用 eksctl 将第二个卷挂载到您的 EC2 实例，您可以使用具有以下配置的节点组：

```
managedNodeGroups:
  - name: al2-workers
    amiFamily: AmazonLinux2
    desiredCapacity: 2
    volumeSize: 80
    additionalVolumes:
      - volumeName: '/dev/sdz'
        volumeSize: 100
    preBootstrapCommands:
    - |
      "systemctl stop containerd"
      "mkfs -t ext4 /dev/nvme1n1"
      "rm -rf /var/lib/containerd/*"
      "mount /dev/nvme1n1 /var/lib/containerd/"
      "systemctl start containerd"
```

如果你使用 terraform 来配置节点组，请参阅 EKS 蓝图中适用于 terraform 的示例。如果您使用 Karpenter 来配置节点，则可以使用blockDeviceMappings节点用户数据来添加其他卷。

要将 EBS 卷直接挂载到您的容器中，您应该使用 AWS EBS CSI 驱动程序并使用具有存储类的卷。

```
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
```

```
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 4Gi
---
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - name: app
    image: public.ecr.aws/docker/library/nginx
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: ebs-claim
```

## 如果工作负载使用 EBS 卷，请避开 EBS 连接限制较低的实例

EBS 是工作负载获得持久存储的最简单方法之一，但它也存在可扩展性限制。每种实例类型都有可连接的最大数量的 EBS 卷。工作负载需要声明它们应该在哪些实例类型上运行，并限制带有 Kubernetes 污染的单个实例上的副本数量。

## 禁用不必要的磁盘日志记录

不要在生产环境中运行带有调试日志的应用程序，并禁用频繁读写磁盘的日志记录，从而避免不必要的本地日志记录。Journald 是一种本地日志服务，它在内存中保留日志缓冲区，并定期刷新到磁盘。Journald 比 syslog 更受青睐，后者会立即将每行记录到磁盘。禁用 syslog 还可以降低所需的存储总量，并避免需要复杂的日志轮换规则。要禁用 syslog，您可以将以下代码段添加到 cloud-init 配置中：

```
runcmd:
  - [ systemctl, disable, --now, syslog.service ]
```

### 当需要操作系统更新速度时，将实例修补到位

> **Important**
>
> 只有在需要时才应在原地修补实例。Amazon 建议将基础设施视为不可变的，并像应用程序一样彻底测试在较低环境中推广的更新。当无法做到这一点时，本节适用。

在不中断容器化工作负载的情况下，在现有 Linux 主机上安装软件包需要几秒钟。无需封锁、耗尽或更换实例即可安装和验证软件包。

要替换实例，您首先需要创建、验证和分发新的 AMI。该实例需要创建替代实例，并且需要封锁和排空旧实例。然后，需要在新实例上创建工作负载，对其进行验证，然后为所有需要修补的实例重复这些工作负载。在不中断工作负载的情况下安全更换实例需要数小时、数天或数周的时间。

Amazon 建议使用基于自动声明式系统构建、测试和升级的不可变基础架构，但如果您需要快速修补系统，则需要修补现有系统，并在新 AMI 可用时替换它们。由于修补和更换系统之间的时间差异很大，我们建议在需要时使用 AWS Systems Manager Patch Manager 自动修补节点。

在 AMI 更新后，通过修补节点，您可以快速推出安全更新并定期更换实例。如果您使用的操作系统具有只读根文件系统，例如 Flatcar Container Linux 或 Bottlerocket OS，我们建议您使用适用于这些操作系统的更新运算符。Flatcar Linux 更新操作员和 Bottlerocket 更新操作员将重启实例，使节点自动保持最新状态。

## 集群服务

集群服务在 EKS 集群内运行，但它们不是用户工作负载。如果您使用的是 Linux 服务器，则通常需要运行 NTP、syslog 和容器运行时等服务来支持您的工作负载。集群服务类似，支持服务可帮助您实现集群的自动化和运营。在 Kubernetes 中，它们通常在 kube-system 命名空间中运行，有些则以 kube-system 命名空间运行。DaemonSets

集群服务预计会有较长的正常运行时间，并且通常在停机期间和故障排除中起着至关重要的作用。如果核心群集服务不可用，您可能会无法访问有助于恢复或防止中断的数据（例如磁盘利用率高）。它们应在专用计算实例（例如单独的节点组或 AWS Fargate）上运行。这将确保共享实例上的集群服务不会受到可能正在扩展或使用更多资源的工作负载的影响。

### 扩展 CoreDNS

扩展 CoreDNS 有两种主要机制。减少对 CoreDNS 服务的调用次数并增加副本数量。

## 通过降低 ndots 来减少外部查询

ndots 设置指定域名中有多少句点（又名“点”）被认为足以避免查询 DNS。如果您的应用程序的 ndots 设置为 5（默认），并且您从外部域（例如 api.example.com）请求资源（2 个点），则系统将针对/.conf 中定义的每个搜索域查询 CoreDNS 以获取更具体的域。etc/resolv默认情况下，在发出外部请求之前将搜索以下域。

```
api.example.<namespace>.svc.cluster.local
api.example.svc.cluster.local
api.example.cluster.local
api.example.<region>.compute.internal
```

`namespace`和`region`值将替换为您的工作负载命名空间和计算区域。根据您的集群设置，您可能还有其他搜索域。

您可以通过降低工作负载的 nd ots 选项来减少 CoreDNS 的请求数量，或者通过添加尾随来完全限定域名请求。（例如`api.example.com.`）。如果您的工作负载通过 DNS 连接到外部服务，我们建议将 ndots 设置为 2，这样工作负载就不会在集群内进行不必要的集群 DNS 查询。如果工作负载不需要访问集群内部的服务，则可以设置不同的 DNS 服务器和搜索域。

```
spec:
  dnsPolicy: "None"
  dnsConfig:
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

如果您将 ndots 降低到过低的值，或者您要连接的域名不包括足够的特异性（包括尾随字符。），那么 DNS 查找可能会失败。请务必测试此设置将如何影响您的工作负载。

## 水平扩展 CoreDNS

CoreDNS 实例可以通过向部署中添加其他副本来进行扩展。建议您使用 NodeLocal DNS 或集群比例自动扩缩器来扩展 CoreDNS。

NodeLocal DNS 将要求每个节点运行一个实例， DaemonSet这需要集群中的更多计算资源，但它可以避免 DNS 请求失败并缩短集群中 DNS 查询的响应时间。集群比例自动扩缩器将根据集群中的节点或内核数量扩展 CoreDNS。这与请求查询没有直接关系，但根据您的工作负载和集群大小，这可能会很有用。默认的比例比例是为集群中每 256 个内核或 16 个节点添加一个额外的副本，以先发生者为准。

如果使用 CoreDNS EKS 附加组件，请考虑启用自动缩放选项。CoreDNS 自动扩缩器通过监控节点数和 CPU 内核来动态调整 CoreDNS 副本的数量，其公式取最大值（nodes^16）或（CPU cores^256），在需要时立即向上扩展，然后逐渐向下扩展以保持稳定性。

## 垂直扩展 Kubernetes 指标服务器

Kubernetes 指标服务器支持水平和垂直扩展。通过水平扩展 Metrics Server，它将具有很高的可用性，但它不会水平扩展以处理更多的集群指标。将节点和收集的指标添加到集群后，您需要根据指标服务器的建议垂直扩展。

指标服务器将其收集、聚合和提供的数据保存在内存中。随着群集的增长，Metrics Server 存储的数据量也随之增加。在大型集群中，Metrics Server 需要的计算资源将超过默认安装中指定的内存和 CPU 预留量。你可以使用 Vertic al Pod Autoscaler (VPA) 或 Addon Resizer 来扩展 Metrics Server。Addon Resizer 根据工作节点成比例垂直扩展，VPA 基于 CPU 和内存使用率进行扩展。

## CoreDNS lameduck 时长

Pod 使用该`kube-dns`服务进行名称解析。Kubernetes 使用目标 NAT (DNAT) 将`kube-dns`流量从节点重定向到 CoreDNS 后端容器。在扩展 CoreDNS 部署时，会更新节点上的 iptables 规则和链`kube-proxy`，将 DNS 流量重定向到 CoreDNS 容器。向上扩展时传播新的终端节点和缩小规模 CoreDNS 时删除规则可能需要 1 到 10 秒的时间，具体取决于集群的大小。

当 CoreDNS 容器被终止但该节点的 iptables 规则尚未更新时，这种传播延迟可能会导致 DNS 查找失败。在这种情况下，该节点可能会继续向已终止的 CoreDNS 容器发送 DNS 查询。

你可以通过在 CoreDNS 容器中设置 l ameduck 时长来减少 DNS 查询失败。在 lameduck 模式下，CoreDNS 将继续响应机上请求。设置 lameduck 持续时间会延迟 CoreDNS 的关闭过程，从而让节点有时间更新 iptables 规则和链。

我们建议将 CoreDNS lameduck 持续时间设置为 30 秒。

## CoreDNS 就绪情况调查

我们建议使用`/ready`代替CoreDNS的就绪性探测。`/health`

与之前提出的将lameduck持续时间设置为30秒的建议，为节点的iptables规则提供充足的时间在吊舱终止之前进行更新，使用`/ready`而不是用于CoreDNS就绪探测可确保CoreDNS容器在启动时做好充分准备，可以迅速响应DNS请求。`/health`

```
readinessProbe:
```

```
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
```

有关 CoreDNS Ready 插件的更多信息，请参阅 https://coredns.io/plugins/ready/

## 记录和监视代理

日志和监控代理可能会给集群控制平面增加大量负载，因为代理会查询 API 服务器以使用工作负载元数据丰富日志和指标。节点上的代理只能访问本地节点资源以查看容器和进程名称之类的内容。查询 API 服务器可以添加更多详细信息，例如 Kubernetes 部署名称和标签。这对于故障排除非常有用，但不利于扩展。

由于有许多不同的日志和监控选项，因此我们无法为每个提供商展示示例。使用 f luentbi t，我们建议启用 Use_Kubelet 从本地 kubelet 而不是 Kubernetes API 服务器获取元数据，并将其设置为`Kube_Meta_Cache_TTL`一个在可以缓存数据时减少重复调用的数字（例如 60）。

扩展监控和日志记录有两个常规选项：

- 禁用集成
- 采样和过滤

禁用集成通常不是一种选择，因为您会丢失日志元数据。这消除了 API 扩展问题，但它会因为在需要时没有所需的元数据而引入其他问题。

采样和过滤可以减少收集的指标和日志的数量。这将减少对 Kubernetes API 的请求量，并减少收集的指标和日志所需的存储量。降低存储成本将降低整个系统的成本。

配置采样的能力取决于代理软件，可以在不同的摄取点实现。在尽可能靠近代理的地方添加采样很重要，因为这可能是 API 服务器调用发生的地方。请联系您的提供商，了解有关采样支持的更多信息。

如果您使用的是 CloudWatch 和 CloudWatch 日志，则可以使用文档中描述的模式添加代理筛选。

为避免丢失日志和指标，您应将数据发送到能够缓冲数据的系统，以防接收端点出现故障。有了 fluentbit，你可以使用 Amazon Kinesis Data Firehose 来临时保留数据，从而减少最终数据存储位置过载的机会。

# 工作负载

工作负载会影响集群的扩展规模。大量使用 Kubernetes API 的工作负载会限制您在单个集群中可以拥有的工作负载总量，但是您可以更改一些默认值以帮助减少负载。

Kubernetes 集群中的工作负载可以访问与 Kubernetes API 集成的功能（例如 Secrets 和 ServiceAccounts），但是这些功能并不总是必需的，如果不使用这些功能，则应将其禁用。限制工作负载访问和对 Kubernetes 控制平面的依赖将增加你可以在集群中运行的工作负载数量，并通过消除对工作负载的不必要访问和实施最低权限做法来提高集群的安全性。请阅读安全最佳实践，了解更多信息。

## 使用 IPv6 进行容器联网

您无法将 VPC 从 IPv4 过渡到 IPv6，因此在配置集群之前启用 IPv6 非常重要。如果您在 VPC 中启用 IPv6，并不意味着您必须使用它；如果您的 pod 和服务使用 IPv6，您仍然可以路由来自 IPv4 地址的流量。有关更多信息，请参阅 EKS 网络最佳实践。

在集群中使用 IPv6 可以避免一些最常见的集群和工作负载扩展限制。IPv6 可避免 IP 地址耗尽，因为没有 IP 地址可用而无法创建 Pod 和节点。它还提高了每个节点的性能，因为通过减少每个节点的 ENI 连接数量，Pod 可以更快地接收 IP 地址。您可以在 VPC CNI 中使用 IPv4 前缀模式实现类似的节点性能，但仍需要确保 VPC 中有足够的 IP 地址可用。

## 限制每个命名空间的服务数量

一个命名空间中的最大服务数为 5,000，集群中的最大服务数为 10,000。为了帮助组织工作负载和服务、提高性能并避免对命名空间范围内的资源产生级联影响，我们建议将每个命名空间的服务数量限制在 500 个。

使用 kube-proxy 为每个节点创建的 IP 表规则数量会随着集群中服务总数的增加而增长。生成成千上万个 IP 表规则并通过这些规则路由数据包会影响节点的性能并增加网络延迟。

只要每个命名空间的服务数量低于 500，即可创建包含单个应用程序环境的 Kubernetes 命名空间。这将使服务发现足够小，以避免服务发现限制，还可以帮助您避免服务命名冲突。应用程序环境（例如开发、测试、生产）应使用单独的 EKS 集群而不是命名空间。

## 了解 Elastic 负载均衡器配额

创建服务时，请考虑将使用哪种类型的负载平衡（例如网络负载均衡器 (NLB) 或应用程序负载均衡器 (ALB)）。每种负载均衡器类型提供不同的功能和不同的配额。有些默认配额可以调整，但有些配额上限无法更改。要查看您的账户配额和使用情况，请查看 AWS 控制台中的 S ervice Quotas 控制面板。

例如，ALB 的默认目标是 1000。如果您的服务拥有超过 1000 个终端节点，则需要增加配额或将服务拆分到多个 ALB 或使用 Kubernetes Ingress。NLB 的默认目标是 3000，但每个可用区限制为 500 个目标。如果您的集群为 NLB 服务运行的容量超过 500 个，则需要使用多个可用区或请求提高配额限制。

使用与服务耦合的负载均衡器的另一种方法是使用入口控制器。AWS Load Balancer 控制器可以为入口资源创建 ALB，但您可以考虑在集群中运行专用控制器。集群内入口控制器允许您通过在集群内运行反向代理来从单个负载均衡器公开多个 Kubernetes 服务。控制器具有不同的功能，例如支持 Gateway API，根据您的工作负载的数量和规模，这些功能可能会带来好处。

## 使用 Route 53、Global Accelerator 或 CloudFront

要将使用多个负载均衡器的服务作为单个终端节点使用，您需要使用亚马逊 CloudFront、AWS Global A ccelerator 或 A mazon Route 53 将所有负载均衡器作为单个面向客户的终端节点公开。每个选项都有不同的好处，可以根据需要单独使用或一起使用。

Route 53 可以用一个通用名称公开多个负载均衡器，并且可以根据分配的权重向每个负载均衡器发送流量。您可以在文档中阅读有关 DNS 权重的更多信息，也可以在 AWS Loa d Balancer Controller 文档中阅读如何使用 Kubernetes 外部 DNS 控制器实现它们。

全球加速器可以根据请求的 IP 地址将工作负载路由到最近的区域。这对于部署到多个区域的工作负载可能很有用，但它并不能改善到单个区域中单个集群的路由。将 Route 53 与全球加速器结合使用还有其他好处，例如运行状况检查和在可用区不可用时自动进行故障转移。您可以在这篇博客文章中看到在 Route 53 上使用全球加速器的示例。

CloudFront 可以与 Route 53 和全球加速器一起使用，也可以单独使用将流量路由到多个目的地。CloudFront 缓存从原始来源提供的资产，这可能会降低带宽需求，具体取决于您提供的服务。

## 使用 EndpointSlices 代替终端节点

在发现与服务标签匹配的 pod 时，应使用EndpointSlices而不是终端节点。端点是小规模公开服务的简单方法，但是自动扩展或有更新的大型服务会导致 Kubernetes 控制平面上的大量流量。EndpointSlices 有自动分组，可以启用拓扑感知提示之类的东西。

并非所有控制器都 EndpointSlices 默认使用。您应该验证您的控制器设置并在需要时将其启用。对于 AWS Load Balancer 控制器，您应该启用要使用的`--enable-endpoint-slices`可选标志 EndpointSlices。

## 如果可能，请使用不可变和外部机密

kubelet 会保留当前密钥和密钥值的缓存，这些密钥用于该节点上 Pod 的卷中。kubelet 会监视秘密以检测变化。随着集群的扩展，监视数量的增加可能会对 API 服务器性能产生负面影响。

有两种策略可以减少 Secrets 上的观看次数：

- 对于不需要访问 Kubernetes 资源的应用程序，你可以通过设置 automount: false 来禁用自动挂载的服务账户密钥 ServiceAccountToken
- 如果您的应用程序的密钥是静态的，并且将来不会被修改，请将该密钥标记为不可变。kubelet 不维护 API 监视不可变的机密。

要禁用自动将服务帐户挂载到 pod，可以在工作负载中使用以下设置。如果特定工作负载需要服务帐号，则可以覆盖这些设置。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app
automountServiceAccountToken: true
```

在集群中的密钥数量超过 10,000 个限制之前对其进行监控。您可以使用以下命令查看集群中的密钥总数。您应该通过集群监控工具监控此限制。

```
kubectl get secrets -A | wc -l
```

您应该设置监控，以便在达到此限制之前提醒集群管理员。考虑使用外部机密管理选项，例如带有 Secrets St ore CSI 驱动程序的 AWS 密钥管理服务 (AWS KMS) 或 Hashicorp Vault。

## 限制部署历史记录

创建、更新或删除 Pod 时可能会很慢，因为集群中仍会跟踪旧对象。你可以减少部署以清理较旧revisionHistoryLimit的部署 ReplicaSets ，这将降低到 Kubernetes 控制器管理器跟踪的对象总数。10 中部署的默认历史记录限制。

如果您的集群通过 CronJobs 或其他机制创建了大量任务对象，则应使用该ttlSecondsAfterFinished设置自动清理集群中的旧 pod。这将在指定时间后从任务历史记录中删除成功执行的作业。

## 默认情况下禁用 “ServiceLinks 启用”

当 Pod 在节点上运行时，kubelet 会为每个活动服务添加一组环境变量。Linux 进程的环境大小有上限，如果命名空间中有太多服务，则可以达到该大小。每个命名空间的服务数量不应超过 5,000。此后，服务环境变量的数量超过了 shell 限制，导致 Pod 在启动时崩溃。

Pod 不应使用服务环境变量进行服务发现还有其他原因。环境变量名称冲突、泄漏的服务名和环境总大小不一而足。您应该使用 CoreDNS 来发现服务端点。

## 限制每个资源的动态准入 webhook

动态准入 Webhook 包括准入 webhook 和变异 Webhook。它们是 API 端点，不是 Kubernetes 控制平面的一部分，当资源发送到 Kubernetes API 时，它们会按顺序调用。每个 webhook 的默认超时时间为 10 秒，如果您有多个 webhook 或其中任何一个 webhook 超时，则可以增加 API 请求所花费的时间。

确保您的 webhook 具有高可用性（尤其是在可用区事件期间），并正确设置 Fa ilurePolicy 以拒绝资源或忽略故障。通过允许--dry-run kubectl 命令绕过 webhook，不要在不需要时调用 webhook。

```
apiVersion: admission.k8s.io/v1
kind: AdmissionReview
request:
  dryRun: False
```

改变 webhook 可以频繁地连续修改资源。如果你有 5 个变异 webhook 并部署 50 个资源，etcd 将存储每种资源的所有版本，直到每隔 5 分钟进行一次压缩，以移除修改后的资源的旧版本。在这种情况下，当 etcd 删除被取代的资源时，将从 etcd 中删除 200 个资源版本，根据资源的大小，可能会在 etcd 主机上占用大量空间，直到每 15 分钟进行一次碎片整理。

这种碎片整理可能会导致 etcd 暂停，这可能会对 Kubernetes API 和控制器产生其他影响。应避免频繁修改大型资源或快速连续修改数百个资源。

## 比较多个集群中的工作负载

如果您有两个集群的性能应该相似但却没有，请尝试比较指标以找出原因。

例如，比较集群延迟是一个常见问题。这通常是由于 API 请求量的差异造成的。你可以运行以下 CloudWatch LogInsight 查询来了解其中的区别。

```
filter @logStream like "kube-apiserver-audit"
```

```
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, userAgent, verb, responseStatus.code
| sort cnt desc
| limit 1000
```

您可以添加其他过滤器来缩小范围。例如，将重点放在来自的所有列表请求上foo。

```
filter @logStream like "kube-apiserver-audit"
| filter verb = "list"
| filter user.username like "foo"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, responseStatus.code
| sort cnt desc
| limit 1000
```

# Kubernetes 缩放理论

## 节点与流失率

通常，当我们讨论 Kubernetes 的可扩展性时，我们会根据单个集群中有多少节点来讨论这个问题。有趣的是，这很少是理解可扩展性的最有用的指标。例如，拥有大量但固定数量的 pod 的 5,000 个节点的集群在初始设置后不会给控制平面带来太大的压力。但是，如果我们选择一个 1,000 个节点的集群，并尝试在不到一分钟的时间内创建 10,000 个短暂的工作岗位，那将给控制平面带来巨大的持续压力。

仅仅使用节点数量来理解缩放可能会产生误导。最好从特定时间段内发生的变化率来考虑（让我们使用 5 分钟的时间间隔进行讨论，因为这是 Prometheus 查询通常默认使用的）。让我们探讨一下为什么用变化率来描述问题可以让我们更好地了解要调整什么才能达到我们想要的规模。

## 每秒查询次数中的思考

Kubernetes 为每个组件（Kubelet、调度器、Kube 控制器管理器和 API 服务器）提供了多种保护机制，以防止 Kubernetes 链中的下一个链接不堪重负。例如，Kubelet 有一个标志，可以按一定速率限制对 API 服务器的调用。这些保护机制通常但并非总是以每秒允许的查询或 QPS 来表达。

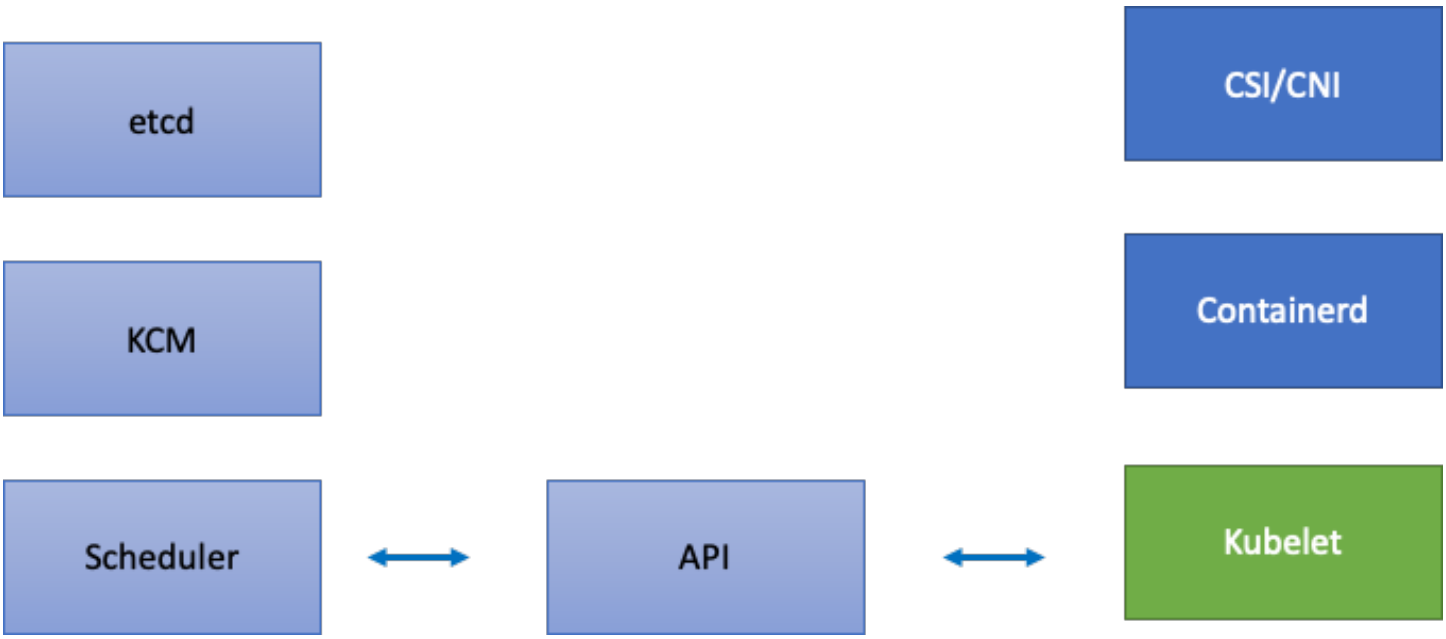
更改这些 QPS 设置时必须格外小心。移除一个瓶颈，例如 Kubelet 上的每秒查询次数，将对其他下游组件产生影响。这可能会而且将会超过一定速度使系统不堪重负，因此，了解和监控服务链的每个部分是成功扩展 Kubernetes 上工作负载的关键。

> **Note**
>
> API 服务器的系统更加复杂，引入了 API 优先级和公平性，我们将单独讨论。

> **Note**
>
> 注意，有些指标看起来很合适，但实际上是在衡量其他指标。举个例子，它只与 Kubelet 中的指标服务器`kubelet_http_inflight_requests`有关，而不是从 Kubelet 向 apiserver 请求的请求数量。这可能会导致我们错误配置 Kubelet 上的 QPS 标志。查询特定 Kubelet 的审计日志将是检查指标的更可靠方法。

## 扩展分布式组件

由于 EKS 是一项托管服务，让我们将 Kubernetes 组件分为两类：AWS 托管组件，包括 etcd、Kube 控制器管理器和调度器（在图表的左侧），以及客户可配置的组件，例如 Kubelet、容器运行时以及调用 AWS API 的各种运算符，例如网络和存储驱动程序（在图的右侧）。尽管 API 服务器由 AWS 管理，但我们仍将其置于中间，因为 API 优先级和公平性设置可以由客户配置。



# 上游和下游瓶颈

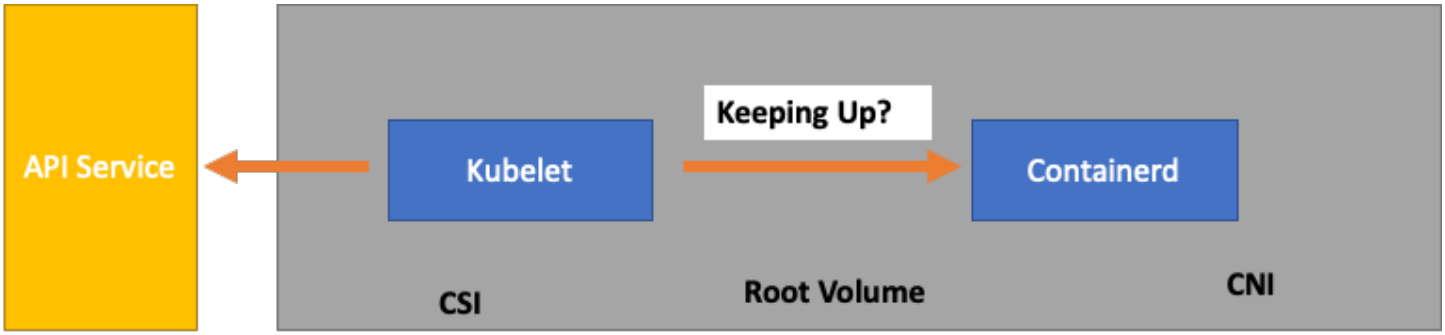
当我们监控每项服务时，重要的是要双向查看指标，以寻找瓶颈。让我们以 Kubelet 为例，学习如何做到这一点。Kubelet 会与 API 服务器和容器运行时进行交谈；我们需要如何以及如何监控哪些组件来检测任何一个组件是否遇到问题？

## 每个节点有多少 Pod

当我们查看缩放数字（例如一个节点上可以运行多少个 Pod）时，我们可以按面值计算上游支持的每个节点 110 个 pod。

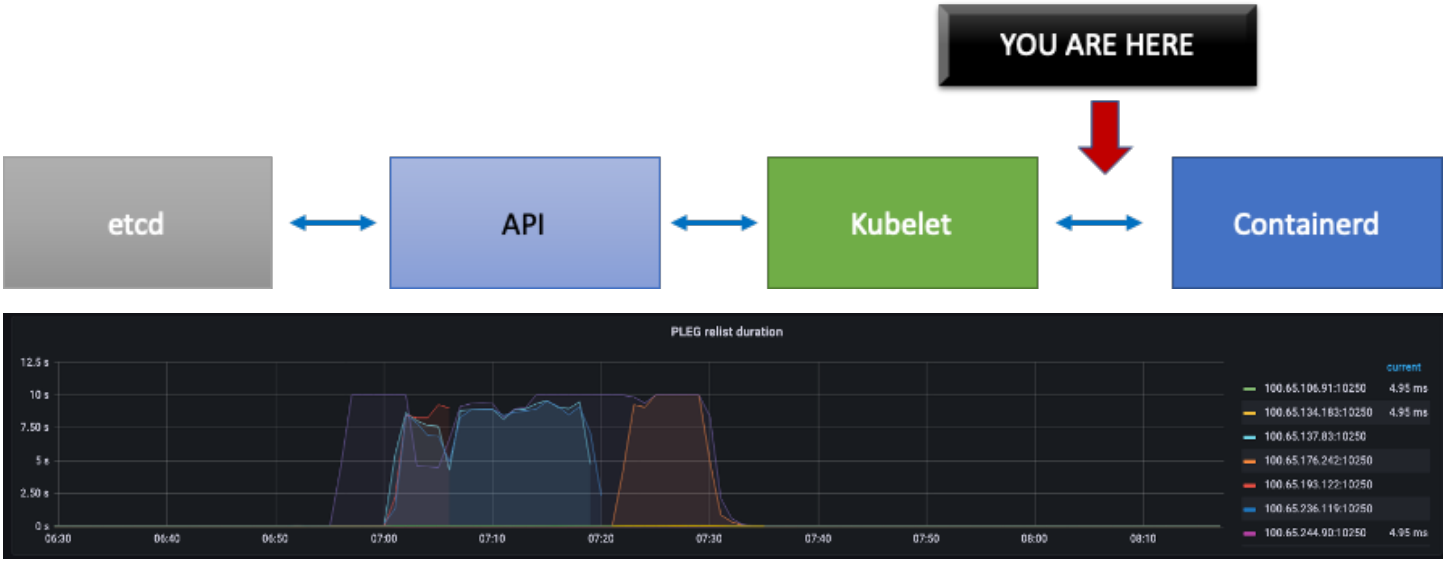
> **Note**
>
> https://kubernetes.io/docs/setup/best-practices/cluster-large/

但是，您的工作负载可能比在上游的可扩展性测试中测试的要复杂得多。为了确保我们可以为要在生产环境中运行的 pod 数量提供服务，让我们确保 Kubelet “跟上” 了 Containerd 运行时的步伐。




简而言之，Kubelet 从容器运行时（在我们的例子中为 Containerd）获取 pod 的状态。如果我们有太多的 pod 过快地改变状态会怎样？ 如果更改率太高，[对容器运行时] 的请求可能会超时。

> **Note**
>
> Kubernetes 在不断发展，这个子系统目前正在发生变化。 https://github.com/kubernetes/enhancements/issues/3386




在上图中，我们看到一条扁线，表示我们刚刚达到了 pod 生命周期事件生成持续时间指标的超时值。如果你想在自己的集群中看到这个，你可以使用以下 PromQL 语法。

```
increase(kubelet_pleg_relist_duration_seconds_bucket{instance="$instance"}
[$__rate_interval])
```

如果我们目睹了这种超时行为，我们就知道我们已经将节点推到了其所能达到的极限。在继续操作之前，我们需要先修复超时的原因。这可以通过减少每个节点的 pod 数量或寻找可能导致大量重试（从而影响流失率）的错误来实现。重要的一点是，与使用固定数字相比，指标是了解节点是否能够处理分配的 pod 流失率的最佳方式。

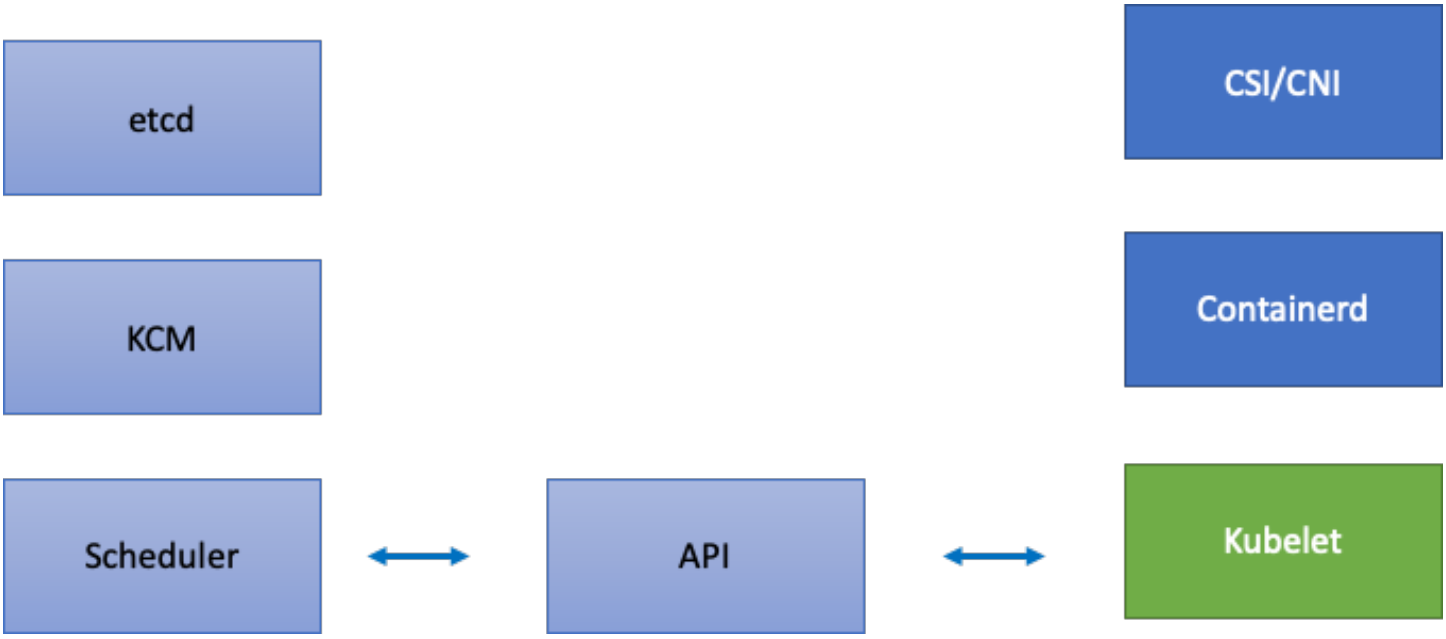
## 按指标缩放

虽然使用指标来优化系统的概念是一个古老的概念，但在人们开始他们的 Kubernetes 之旅时，它经常被忽视。我们没有关注特定的数字（即每个节点 110 个 pod），而是将精力集中在寻找有助于我们发现系统瓶颈的指标上。了解这些指标的正确阈值可以使我们对系统配置最优有信心。

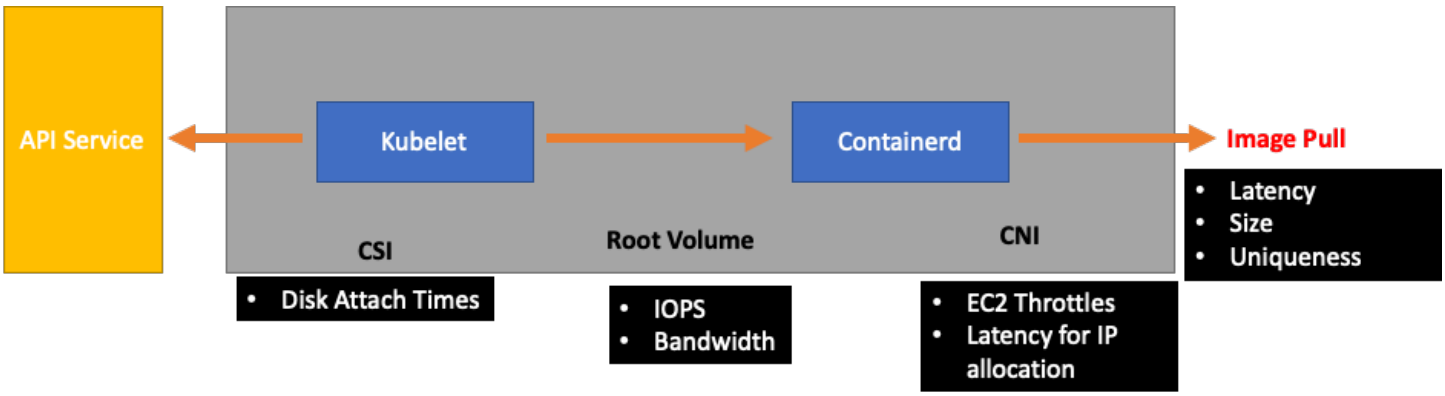
### 变化的影响

可能给我们带来麻烦的一种常见模式是将注意力集中在第一个看起来可疑的指标或日志错误上。当我们看到 Kubelet 早些时候超时时，我们可以随机尝试一些东西，比如提高允许 Kubelet 发送的每秒速率等。但是，明智的做法是查看我们首先发现的错误下游所有内容的全貌。每一次更改都要有目的并以数据为后盾。

Kubelet 的下游将是 Containerd 运行时（容器错误），DaemonSets 例如与 EC2 API 通信的存储驱动程序 (CSI) 和网络驱动程序 (CNI) 等。




让我们继续我们之前的 Kubelet 跟不上运行时间的示例。在很多情况下，我们可以将一个节点装得如此密集，以至于它会触发错误。




在为我们的工作负载设计合适的节点大小时，这些信号很容易被忽视，可能会给系统带来不必要的压力，从而限制我们的规模和性能。

## 不必要的错误的代价

Kubernetes 控制器擅长在出现错误情况时进行重试，但这是有代价的。这些重试可能会增加 Kube 控制器管理器等组件的压力。监控此类错误是规模测试的重要组成部分。

当发生的错误较少时，就更容易发现系统中的问题。通过在重大操作（例如升级）之前定期确保我们的集群没有错误，我们可以简化发生不可预见的事件时的故障排除日志。

扩大我们的视野

在拥有 1,000 个节点的大型集群中，我们不想单独寻找瓶颈。在 PromQL 中，我们可以使用名为 topk 的函数找到数据集中的最高值；K 是一个变量，我们放置了我们想要的项目数。在这里，我们使用三个节点来了解集群中的所有 Kubelet 是否都已饱和。到目前为止，我们一直在研究延迟，现在让我们看看 Kubelet 是否在丢弃事件。

```
topk(3, increase(kubelet_pleg_discard_events{}[$__rate_interval]))
```

分解这句话。

- 我们使用 Grafana `$__rate_interval` 变量来确保它获得所需的四个样本。这就绕过了使用简单变量进行监控的复杂话题。
- `topk`只给我们最好的结果，数字 3 将这些结果限制为三个。对于集群范围的指标，这是一个有用的函数。
- `{}`告诉我们没有过滤器，通常你会输入任何抓取规则的任务名称，但是由于这些名称各不相同，我们将其留空。

将问题一分为二

为了解决系统的瓶颈，我们将采取一种方法来寻找一个指标，该指标可以向我们显示上游或下游存在问题，因为这使我们能够将问题一分为二。这也将是我们如何显示指标数据的核心原则。

开始这个过程的一个不错的地方是 API 服务器，因为它允许我们查看客户端应用程序或控制平面是否存在问题。

# 控制平面监控

## API 服务器

在查看我们的 API 服务器时，请务必记住，其功能之一是限制入站请求以防止控制平面过载。看似在 API 服务器级别的瓶颈实际上可能是保护它免受更严重问题的侵害。我们需要考虑增加通过系统的请求量的利弊。为了确定是否应增加 API 服务器的值，以下是我们需要注意的事项的小示例：

1. 在系统中传输的请求的延迟是多少？
2. 那是 API 服务器本身的延迟，还是像 etcd 这样的 “下游”？
3. API 服务器队列深度是造成这种延迟的一个因素吗？

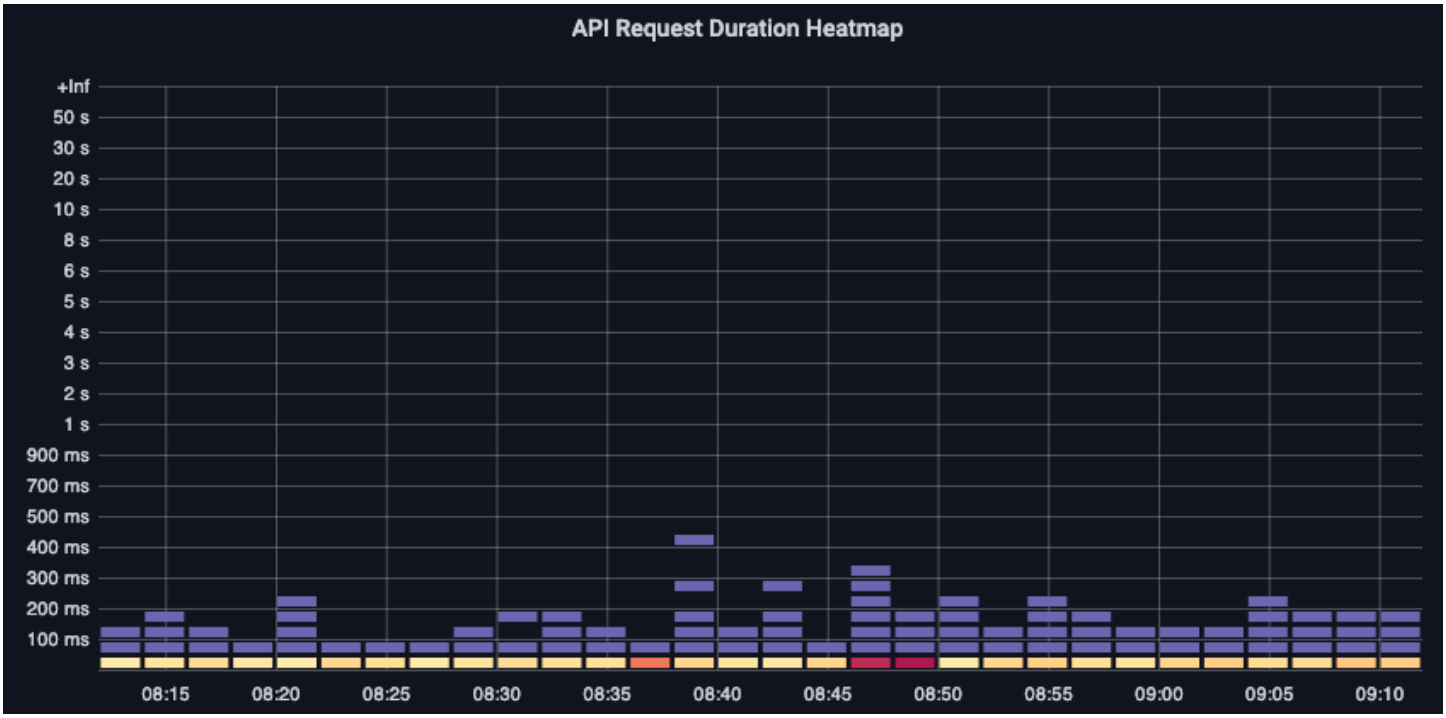
4. API 优先级和公平性 (APF) 队列是否针对我们想要的 API 调用模式设置正确？

## 问题出在哪里？

首先，我们可以使用 API 延迟指标来深入了解 API 服务器为请求提供服务所花费的时间。让我们使用下面的 PromQL 和 Grafana 热图来显示这些数据。

```
max(increase(apiserver_request_duration_seconds_bucket{subresource!
="status",subresource!="token",subresource!="scale",subresource!="/
healthz",subresource!="binding",subresource!="proxy",verb!="WATCH"}[$__rate_interval]))
 by (le)
```

> **Note**
>
> 要详细了解如何使用本文中使用的 API 控制面板监控 API 服务器，请参阅以下博客



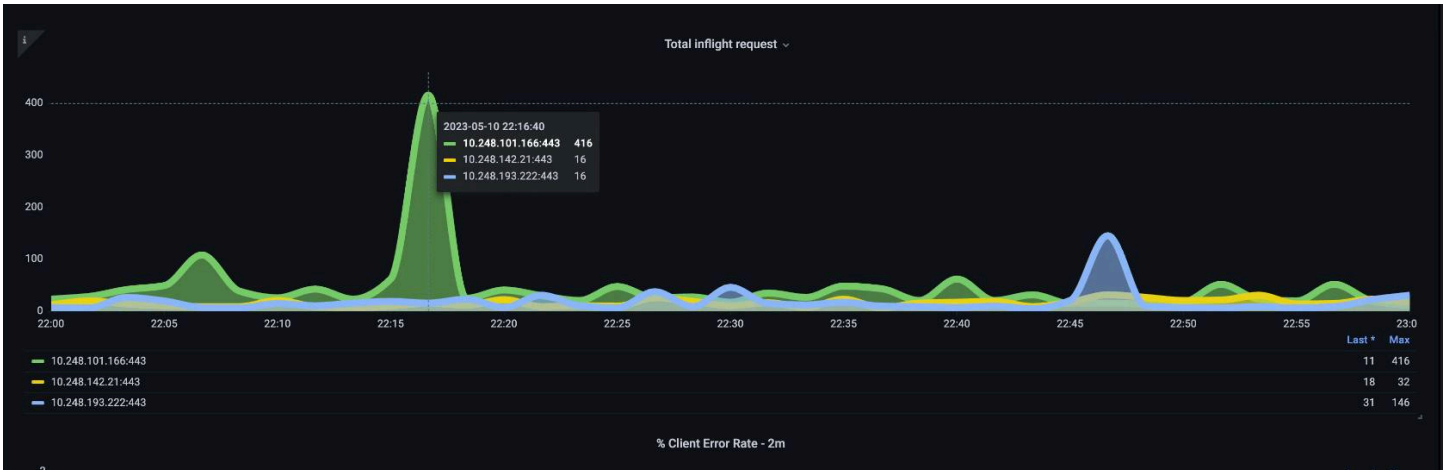

这些请求都在一秒钟以内，这很好地表明控制平面正在及时处理请求。但是，如果不是这样呢？

我们在上述 API 请求持续时间中使用的格式是热图。热图格式的好处在于，它可以告诉我们默认情况下 API 的超时值（60 秒）。但是，我们真正需要知道的是，在达到超时阈值之前，这个值应该在什么阈值上引起关注。有关可接受阈值的粗略指导，我们可以使用上游 Kubernetes SLO，可以在这里找到

> **Note**
>
> 注意到这个语句上的 max 函数了吗？ 使用聚合多台服务器（EKS 上默认为两台 API 服务器）的指标时，重要的是不要将这些服务器放在一起进行平均值。

## 非对称流量模式

如果一个 API 服务器 [pod] 加载得很轻，而另一个负载很重怎么办？ 如果我们将这两个数字相加求平均值，我们可能会误解正在发生的事情。例如，这里有三个 API 服务器，但所有负载都在其中一个 API 服务器上。通常，在投资规模和性能问题时，任何具有多台服务器（例如etcd和API服务器）的东西都应进行细分。



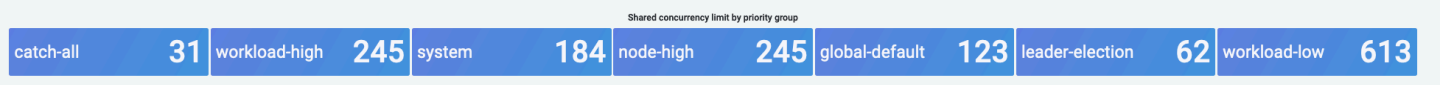

随着转向 API 优先级和公平性，系统上的请求总数只是检查 API 服务器是否超额订阅的一个因素。由于系统现在可以处理一系列队列，因此我们必须查看这些队列中是否有任何队列已满，以及该队列的流量是否被丢弃。

让我们用以下查询来看看这些队列：

```
max without(instance)(apiserver_flowcontrol_nominal_limit_seats{})
```

> **Note**
>
> 有关 API A&F 工作原理的更多信息，请参阅以下[最佳实践指南]

在这里，我们可以看到集群中默认出现的七个不同的优先级组




接下来，我们想看看该优先级组中使用了多少百分比，这样我们就可以了解某个优先级是否已饱和。将请求限制在工作负载低级别可能是可取的，但是降低领导者选举级别则不是。

API 优先级与公平性 (APF) 系统有许多复杂的选项，其中一些选项可能会产生意想不到的后果。我们在现场看到的一个常见问题是将队列深度增加到开始增加不必要的延迟的程度。我们可以使用`apiserver_flowcontrol_current_inqueue_request`指标来监控这个问题。我们可以使用.来检查掉落情况`apiserver_flowcontrol_rejected_requests_total`。如果任何存储桶超过其并发度，则这些指标将为非零值。




增加队列深度会使 API 服务器成为延迟的重要来源，因此应谨慎行事。我们建议谨慎对待创建的队列数量。例如，EKS 系统上的共享数量为 600，如果我们创建的队列太多，这可能会减少需要吞吐量的重要队列（例如领导者选举队列或系统队列）中的份额。创建过多的额外队列会使正确调整这些队列的大小变得更加困难。

为了专注于您可以在APF中进行的简单而有影响力的更改，我们只需从未充分利用的存储桶中提取份额，然后增加处于最大使用量的存储桶的大小。通过在这些桶之间智能地重新分配股份，您可以降低下跌的可能性。

有关更多信息，请访问《EKS 最佳实践指南》中的 API 优先级和公平性设置。

## API 与 etcd 延迟

我们如何使用 API 服务器 metrics/logs 的来确定 API 服务器是否存在问题、API 服务器问题，或者两者兼而有之。 upstream/downstream 为了更好地理解这一点，让我们来看看 API Server 和 etcd 是如何关联的，以及对错误的系统进行故障排除有多容易。

在下图中，我们看到了 API 服务器延迟，但我们也看到其中大部分延迟与 etcd 服务器相关，因为图表中的条形显示了 etcd 级别的大部分延迟。如果同时有 15 秒的 etcd 延迟，则有 20 秒的 API 服务器延迟，则大部分延迟实际上处于 etcd 级别。

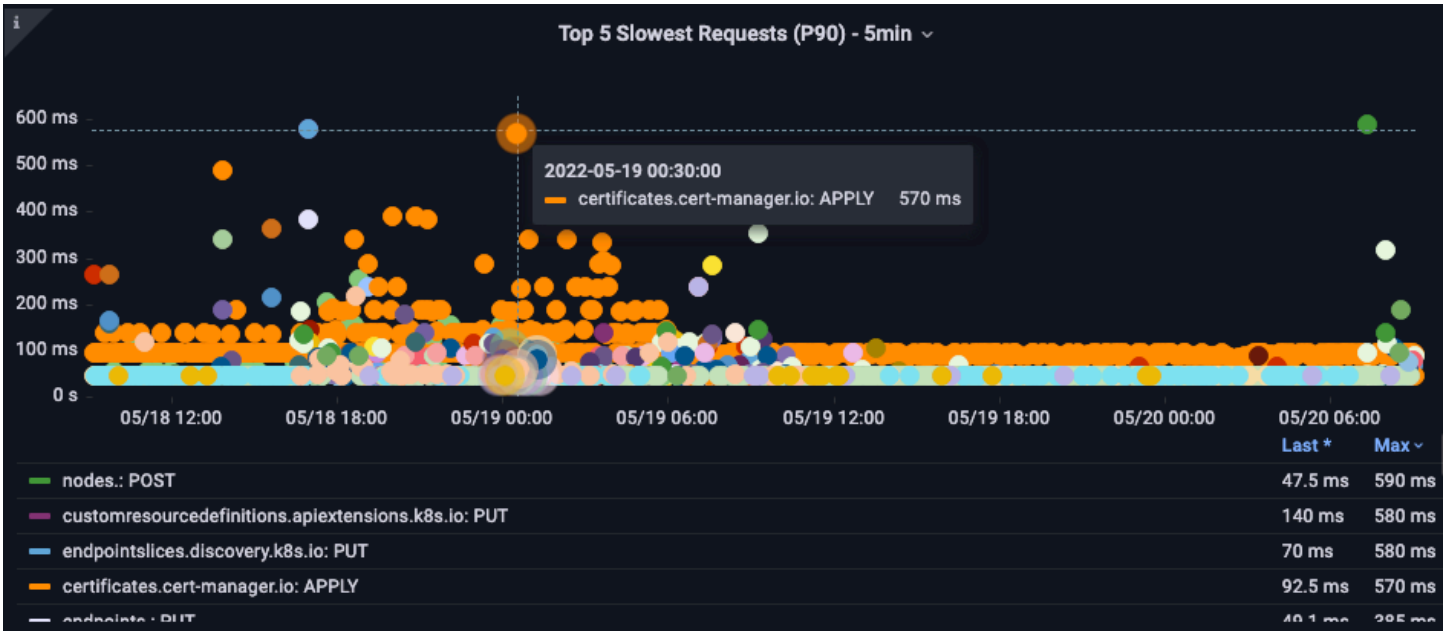
通过查看整个流程，我们发现明智的做法是不要只关注 API 服务器，还要寻找表明 etcd 受到胁迫的信号（即缓慢的应用计数器增加）。只需一眼就能快速移动到正确的问题区域是仪表板的强大之处。

> **Note**
>
> 部分中的仪表板可在以下网址找到 https://github.com/RiskyAdventure/Troubleshooting-Dashboards/blob/main/api-troubleshooter.json




## 控制平面与客户端问题

在此图表中，我们正在寻找在此期间完成时间最长的 API 调用。在本例中，我们看到自定义资源 (CRD) 正在调用 APPLY 函数，这是 05:40 时间范围内最潜在的调用。




有了这些数据，我们就可以在该时间段内使用 Ad-Hoc PromQL 或 CloudWatch Insights 查询从审计日志中提取列表请求，以查看这可能是哪个应用程序。

## 通过以下方式寻找来源 CloudWatch

指标最好用于查找我们要查看的问题区域，并缩小问题的时间范围和搜索参数。有了这些数据后，我们想过渡到日志，以获取更详细的时间和错误。为此，我们将使用 Logs Insights 将CloudWatch 日志转换为指标。

例如，为了调查上述问题，我们将使用以下 L CloudWatch ogs Insights 查询来提取 userAgent 和 requestURI，这样我们就可以确定哪个应用程序导致了这种延迟。

> **Note**
>
> 需要使用适当的计数，以免手表 List/Resync 出现正常行为。

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter ispresent(requestURI)
| filter verb = "list"
| parse requestReceivedTimestamp /\d+-\d+-(?<StartDay>\d+)T(?<StartHour>\d+):(?
<StartMinute>\d+):(?<StartSec>\d+).(?<StartMsec>\d+)Z/
| parse stageTimestamp /\d+-\d+-(?<EndDay>\d+)T(?<EndHour>\d+):(?<EndMinute>\d+):(?
<EndSec>\d+).(?<EndMsec>\d+)Z/
| fields (StartHour * 3600 + StartMinute * 60 + StartSec + StartMsec / 1000000) as
 StartTime, (EndHour * 3600 + EndMinute * 60 + EndSec + EndMsec / 1000000) as EndTime,
 (EndTime - StartTime) as DeltaTime
| stats avg(DeltaTime) as AverageDeltaTime, count(*) as CountTime by requestURI,
 userAgent
| filter CountTime >=50
| sort AverageDeltaTime desc
```

通过此查询，我们发现两个不同的代理正在运行大量高延迟列表操作。Splunk 和 CloudWatch 特工。有了这些数据，我们就可以决定移除、更新这个控制器，或者用另一个项目替换这个控制器。

| requestURI | AverageDeltaTime | AverageD | CountTime |
|---|---|---|---|
| **/api/v1/pods** | rest-client/2.1.0 (linux-gnu x86_64) ruby/2.5.5p157 | 9.1395 | 87 |
| **/api/v1/pods?limit=500&resourceVersion=0** | amazon-cloudwatch-agent/v0.0.0 (linux/amd64) kubernetes/$Format | 5.3976 | 929 |

> **Note**
>
> 有关此主题的更多详细信息，请参阅以下博客

# 调度器

由于 EKS 控制平面实例是在单独的 AWS 账户中运行的，因此我们将无法抓取这些组件来获取指标（API 服务器是个例外）。但是，由于我们可以访问这些组件的审计日志，因此我们可以将这些日志转换为指标，以查看是否有任何子系统导致了扩展瓶颈。让我们使用 Logs CloudWatch Insights 来查看调度器队列中有多少未调度的 pod。

## 调度器日志中未调度的 pod

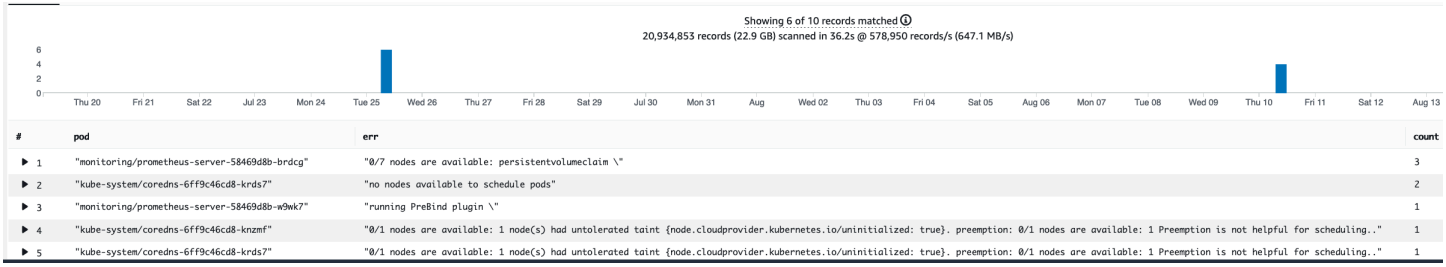
如果我们可以直接在自我管理的 Kubernetes（例如 Kops）上抓取调度器指标，我们将使用以下 PromQL 来了解调度器待办事项。

```
max without(instance)(scheduler_pending_pods)
```

由于我们无法在 EKS 中访问上述指标，因此我们将使用以下 Lo CloudWatch gs Insights 查询来查看在特定时间段内有多少 pod 无法取消调度，从而查看积压情况。然后，我们可以进一步研究高峰时段的消息，以了解瓶颈的本质。例如，节点的旋转速度不够快，或者调度器本身的速率限制器。

```
fields timestamp, pod, err, @message
| filter @logStream like "scheduler"
| filter @message like "Unable to schedule pod"
| parse @message  /^.(?<date>\d{4})\s+(?<timestamp>\d+:\d+:\d+\.\d+)\s+\S*\s+\S+\]\s
\"(.*?)\"\s+pod=(?<pod>\"(.*?)\")\s+err=(?<err>\"(.*?)\")/
| count(*) as count by pod, err
| sort count desc
```

在这里，我们看到了调度程序的错误，说由于存储 PVC 不可用，Pod 没有部署。




| # | pod | err | count |
|---|---|---|---|
| ▶ 1 | "monitoring/prometheus-server-58469d8b-brdcg" | "0/7 nodes are available: persistentvolumeclaim \" | 3 |
| ▶ 2 | "kube-system/coredns-6ff9c46cd8-krds7" | "no nodes available to schedule pods" | 2 |
| ▶ 3 | "monitoring/prometheus-server-58469d8b-w9wk7" | "running PreBind plugin \" | 1 |
| ▶ 4 | "kube-system/coredns-6ff9c46cd8-knzmf" | "0/1 nodes are available: 1 node(s) had untolerated taint {node.cloudprovider.kubernetes.io/uninitialized: true}. preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.." | 1 |
| ▶ 5 | "kube-system/coredns-6ff9c46cd8-krds7" | "0/1 nodes are available: 1 node(s) had untolerated taint {node.cloudprovider.kubernetes.io/uninitialized: true}. preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.." | 1 |

> **Note**
>
> 必须在控制平面上开启审计日志才能启用此功能。限制日志保留也是一种最佳做法，以免随着时间的推移不必要地增加成本。使用下面的 EKSCTL 工具开启所有日志功能的示例。

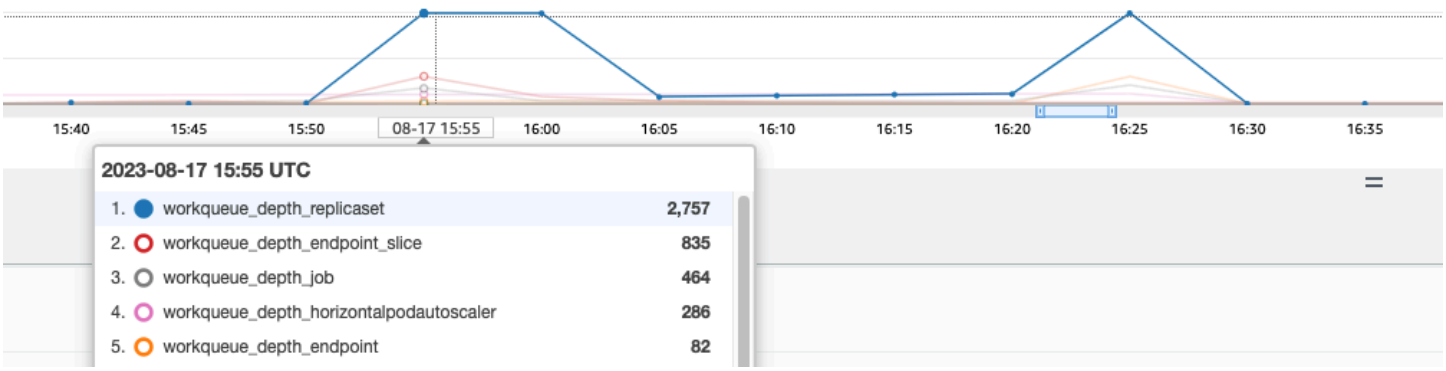
```
cloudWatch:
  clusterLogging:
    enableTypes: ["*"]
    logRetentionInDays: 10
```

## Kube 控制器管理器

与所有其他控制器一样，Kube 控制器管理器对一次可以执行的操作数量有限制。让我们通过查看可以在其中设置这些参数的 KOPS 配置来回顾一下其中一些标志是什么。

```
  kubeControllerManager:
    concurrentEndpointSyncs: 5
    concurrentReplicasetSyncs: 5
    concurrentNamespaceSyncs: 10
    concurrentServiceaccountTokenSyncs: 5
    concurrentServiceSyncs: 5
    concurrentResourceQuotaSyncs: 5
    concurrentGcSyncs: 20
    kubeAPIBurst: 30
    kubeAPIQPS: "20"
```

这些控制器的队列在集群流失率高时会被填满。在本例中，我们看到 replicaset 集控制器的队列中有大量待办事项。




我们有两种不同的方法可以解决这种情况。如果运行自我管理，我们可以简单地增加并发 goroutine，但是这会在 KCM 中处理更多数据，从而对 etcd 产生影响。另一种选择是减少部署中使用的`.spec.revisionHistoryLimit`副本集对象的数量，以减少我们可以回滚的副本集对象的数量，从而减轻该控制器的压力。

```
spec:
```

```
  revisionHistoryLimit: 2
```

可以调整或关闭 Kubernetes 的其他 Kubernetes 功能，以减轻高流失率系统中的压力。例如，如果我们 pod 中的应用程序不需要直接与 k8s API 通信，那么关闭这些 pod 中的投影密将减少负载。ServiceaccountTokenSyncs如果可能，这是解决此类问题的更理想方法。

```
kind: Pod
spec:
  automountServiceAccountToken: false
```

在我们无法访问指标的系统中，我们可以再次查看日志来检测争用情况。如果我们想查看每个控制器或汇总级别上正在处理的请求数量，我们可以使用以下 L CloudWatch ogs Insights 查询。

## KCM 处理的总量

```
# Query to count API qps coming from kube-controller-manager, split by controller type.
# If you're seeing values close to 20/sec for any particular controller, it's most
 likely seeing client-side API throttling.
fields @timestamp, @logStream, @message
| filter @logStream like /kube-apiserver-audit/
| filter userAgent like /kube-controller-manager/
# Exclude lease-related calls (not counted under kcm qps)
| filter requestURI not like "apis/coordination.k8s.io/v1/namespaces/kube-system/
leases/kube-controller-manager"
# Exclude API discovery calls (not counted under kcm qps)
| filter requestURI not like "?timeout=32s"
# Exclude watch calls (not counted under kcm qps)
| filter verb != "watch"
# If you want to get counts of API calls coming from a specific controller, uncomment
 the appropriate line below:
# | filter user.username like "system:serviceaccount:kube-system:job-controller"
# | filter user.username like "system:serviceaccount:kube-system:cronjob-controller"
# | filter user.username like "system:serviceaccount:kube-system:deployment-controller"
# | filter user.username like "system:serviceaccount:kube-system:replicaset-controller"
# | filter user.username like "system:serviceaccount:kube-system:horizontal-pod-
autoscaler"
# | filter user.username like "system:serviceaccount:kube-system:persistent-volume-
binder"
# | filter user.username like "system:serviceaccount:kube-system:endpointslice-
controller"
# | filter user.username like "system:serviceaccount:kube-system:endpoint-controller"
```

```
# | filter user.username like "system:serviceaccount:kube-system:generic-garbage-
controller"
| stats count(*) as count by user.username
| sort count desc
```

这里的关键要点是，在研究可扩展性问题时，在进入详细的故障排除阶段之前，要先查看路径中的每个步骤（API、调度程序、KCM 等）。通常在生产环境中，你会发现需要对 Kubernetes 的多个部分进行调整，才能让系统以最佳性能运行。很容易无意中对瓶颈大得多的症状（例如节点超时）进行故障排除。

## ETCD

etcd 使用内存映射文件有效地存储键值对。有一种保护机制可以设置此可用内存空间的大小，通常设置为 2、4 和 8GB 的限制。数据库中较少的对象意味着在更新对象和需要清理旧版本时，etcd 需要做的清理工作更少。这种清理对象的旧版本的过程称为压缩。经过多次压缩操作后，会有一个后续的过程来恢复可用空间空间，称为碎片整理，该过程发生在超过一定阈值或固定时间表上。

我们可以做一些与用户相关的项目来限制 Kubernetes 中对象的数量，从而减少压缩和碎片整理过程的影响。例如，Helm 保持高位`revisionHistoryLimit`。这样可以使系统 ReplicaSets 上的旧对象能够进行回滚。通过将历史记录限制设置为 2，我们可以将对象（比如 ReplicaSets）的数量从十个减少到两个，这反过来又会减少系统的负载。

```
apiVersion: apps/v1
kind: Deployment
spec:
  revisionHistoryLimit: 2
```

从监控的角度来看，如果系统延迟峰值以设定的模式出现，以小时为间隔，则检查此碎片整理过程是否是源头可能会有所帮助。我们可以使用 Logs 来看到 CloudWatch 这一点。

如果您想查看碎片整理 start/end 时间，请使用以下查询：

```
fields @timestamp, @message
| filter @logStream like /etcd-manager/
| filter @message like /defraging|defraged/
| sort @timestamp asc
```

| @message | @logStream |
| --- | --- |
| Oct 6 19:19:52 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:19:52.826Z","caller":"defrag/defrag.go:312","msg":"defraging","… | etcd-manager-i-0101a20d88e74fa5b |
| Oct 6 19:20:28 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:20:28.836Z","caller":"defrag/defrag.go:322","msg":"defraged"} | etcd-manager-i-0101a20d88e74fa5b |
| Oct 6 19:25:57 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:25:57.650Z","caller":"defrag/defrag.go:312","msg":"defraging"… | etcd-manager-i-0e3aa5c2ae801af44 |
| Oct 6 19:26:34 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:26:34.457Z","caller":"defrag/defrag.go:322","msg":"defraged"} | etcd-manager-i-0e3aa5c2ae801af44 |
| Oct 6 19:32:26 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:32:26.592Z","caller":"defrag/defrag.go:312","msg":"defraging",… | etcd-manager-i-025ebb6b463ee0f21 |
| Oct 6 19:33:09 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:33:09.575Z","caller":"defrag/defrag.go:322","msg":"defraged"} | etcd-manager-i-025ebb6b463ee0f21 |

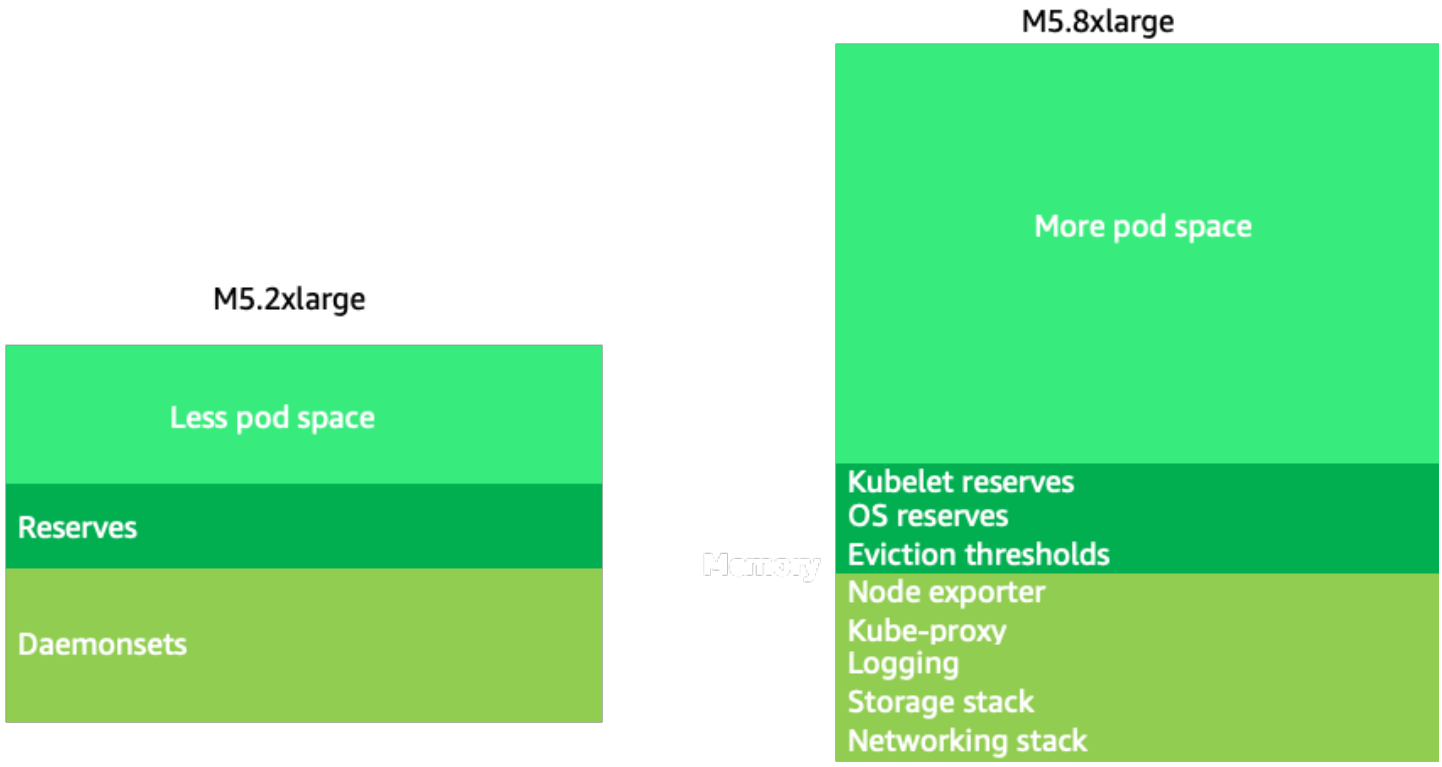
# 节点和工作负载效率

提高工作负载和节点的效率可以减少工作负载和节点， complexity/cost 同时提高性能和规模。在规划这种效率时，有许多因素需要考虑，最容易从权衡角度来考虑，而不是为每个功能设置一个最佳实践设置。让我们在下一节中深入探讨这些权衡。

## 节点选择

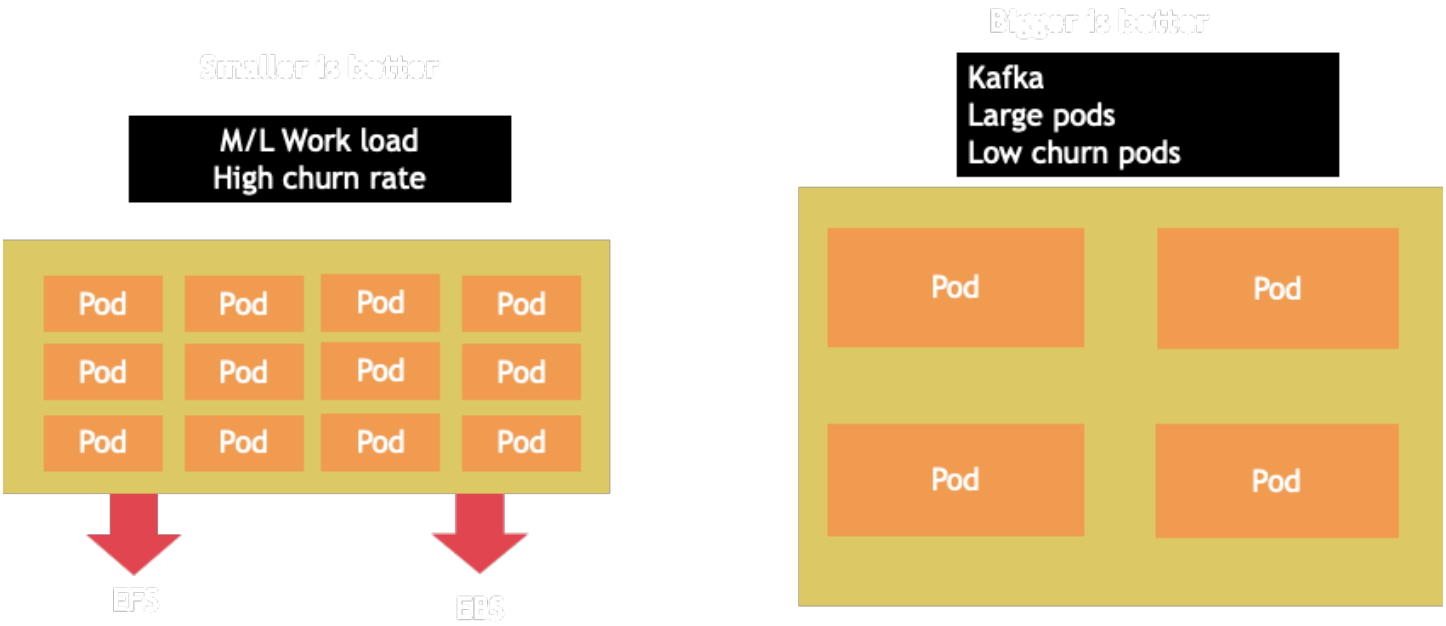
使用稍大一点的节点大小（ 4-12xlarge ）会增加我们用于运行 Pod 的可用空间，因为这样可以减少用于 “开销” （ 例如DaemonSets系统组件的预留空间 ）的节点百分比。在下图中，我们看到了 2xlarge 系统上的可用空间与只有中等数量的 8xlarge 系统的可用空间之间的区别。 DaemonSets

> **Note**
>
> 由于 k8s 通常水平扩展，因此对于大多数应用程序来说，考虑 NUMA 大小节点对性能的影响是没有意义的，因此建议使用低于该节点大小的范围。




较大的节点大小允许我们在每个节点上拥有更高的可用空间百分比。但是，通过在节点上打包太多 pod 以至于会导致错误或使节点饱和，可以将此模型发挥到极致。监控节点饱和度是成功使用更大节点大小的关键。

节点选择很少是一刀切的主张。通常，最好将流失率截然不同的工作负载分成不同的节点组。流失率高的小批量工作负载最好由 4xlarge 系列实例来处理，而 12xlarge 系列可以更好地为像 Kafka 这样的占用 8 个 vCPU 且流失率低的大型应用程序提供服务。




> **Note**
>
> 对于非常大的节点大小，需要考虑的另一个因素是，因为 CGROUPS 不会向容器化应用程序隐藏 vCPU 的总数。动态运行时通常会产生意外数量的操作系统线程，从而造成难以排除故障的延迟。对于这些应用程序，建议使用 CPU 固定。要更深入地探讨话题，请看以下视频 https://www.youtube.com/watch？ v= _kaqG NqtfDy

# 节点 Bin-packing

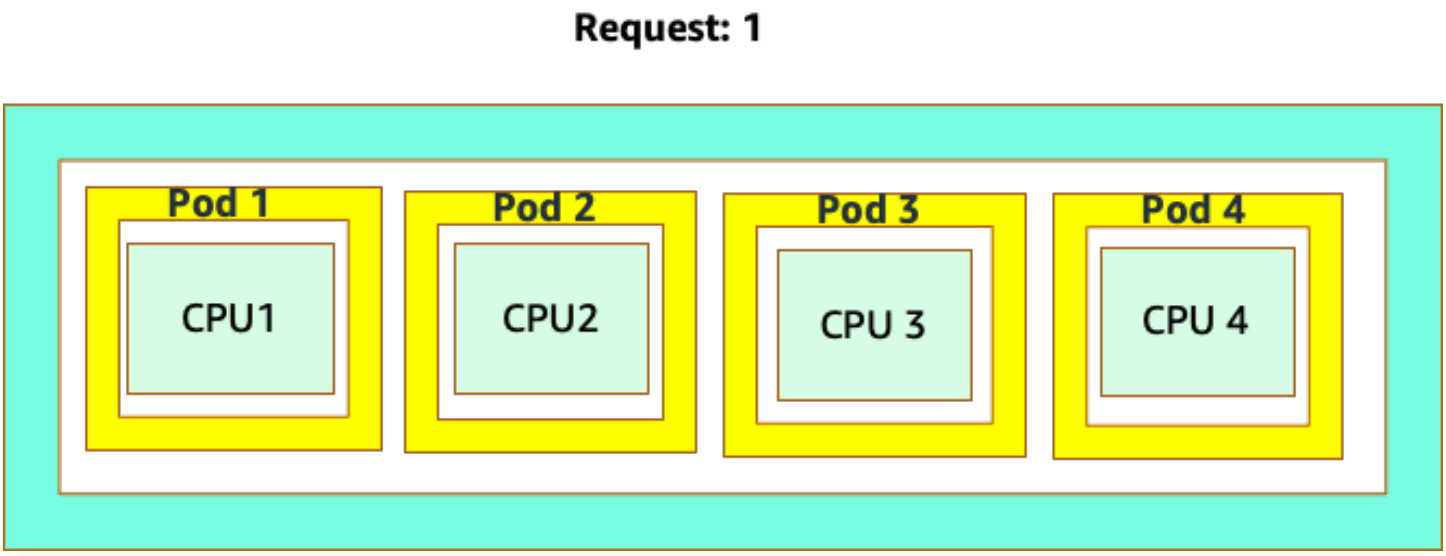
## Kubernetes 与 Linux 规则

在处理 Kubernetes 上的工作负载时，我们需要注意两组规则。Kubernetes 调度器的规则，它使用请求值在节点上调度 Pod，然后在调度 pod 之后会发生什么，这是 Linux 的领域，而不是 Kubernetes 的领域。

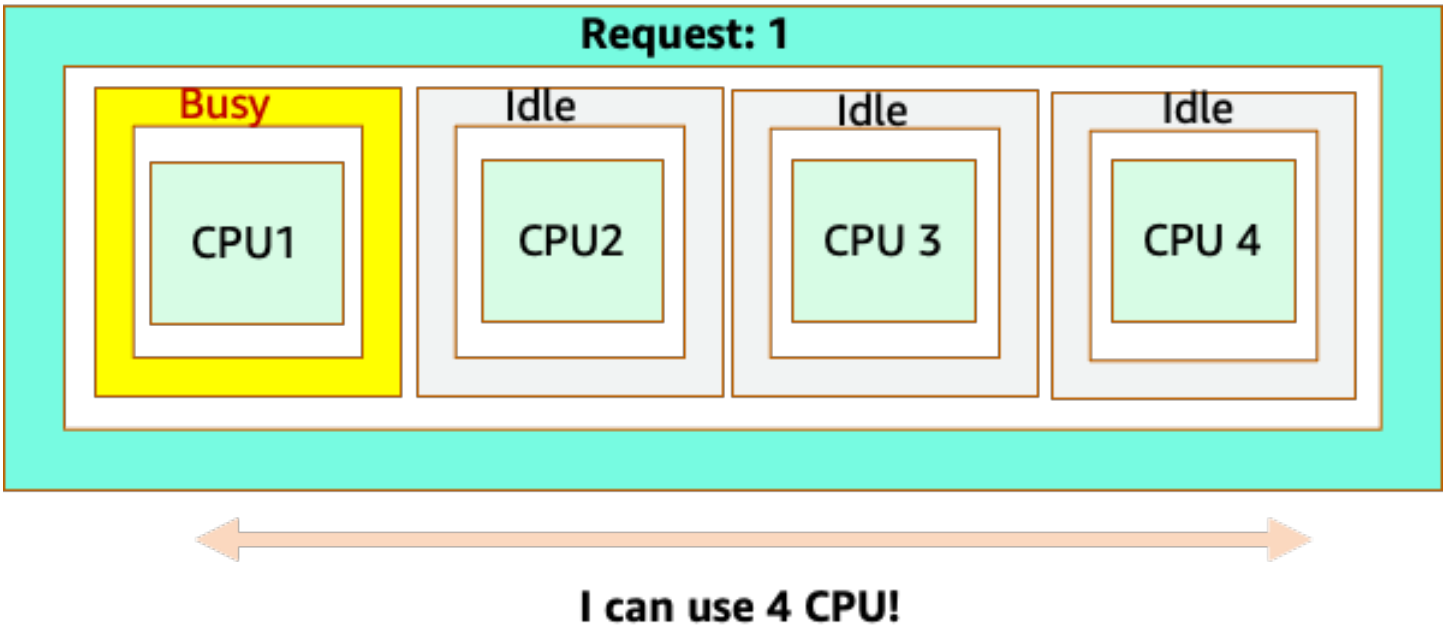
Kubernetes 调度器完成后，一套新的规则取而代之，即 Linux 完全公平的调度器 (CFS)。关键要点是 Linux CFS 没有核心的概念。我们将讨论为什么在内核中思考会导致优化工作负载以实现规模化时出现重大问题。

## 在核心中思考

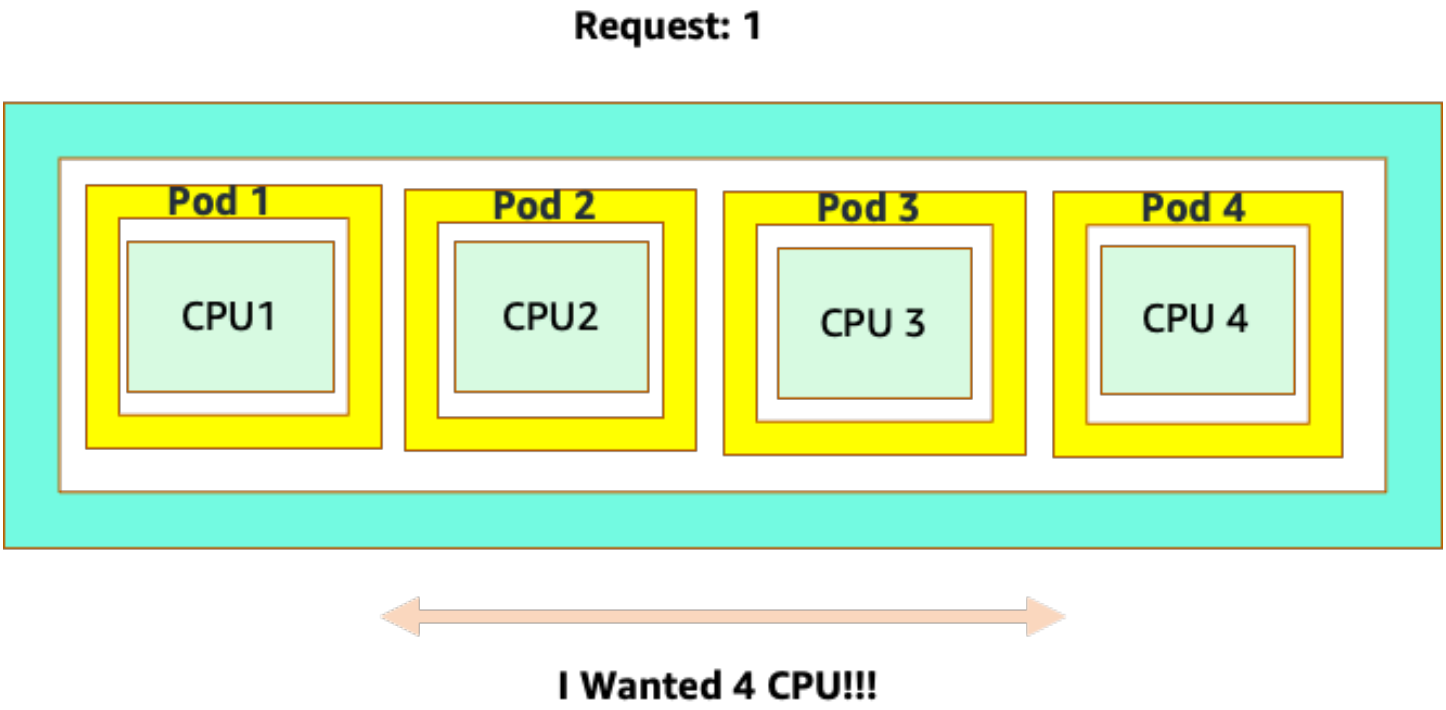
混乱之所以开始，是因为 Kubernetes 调度器确实有内核的概念。从 Kubernetes 调度器的角度来看，如果我们看一个包含 4 个 NGINX Pod 的节点，每个节点都有一个核心集的请求，那么该节点将如下所示。




但是，让我们做一个思想实验，看看从 Linux CFS 的角度来看，这看起来有何不同。使用 Linux CFS 系统时要记住的最重要的一点是：繁忙的容器 (CGROUPS) 是唯一计入共享系统的容器。在这种情况下，只有第一个容器处于繁忙状态，因此允许它使用节点上的所有 4 个内核。



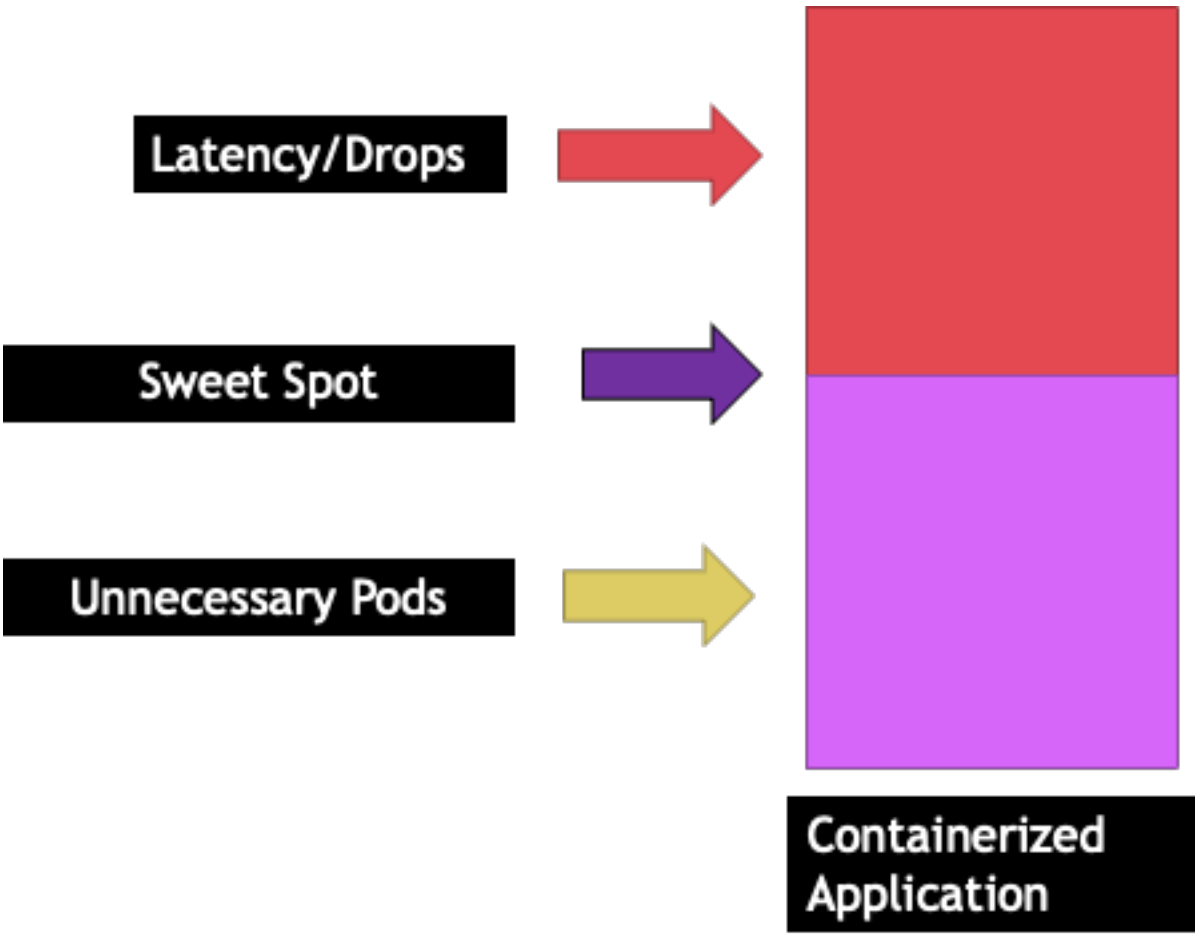
为什么这很重要？ 假设我们在一个开发集群中进行了性能测试，其中 NGINX 应用程序是该节点上唯一繁忙的容器。当我们将应用程序移至生产环境时，会发生以下情况：NGINX 应用程序需要 4 个 vCPU 资源，但是，由于节点上的所有其他 pod 都处于繁忙状态，因此我们的应用程序的性能受到限制。




这种情况会导致我们不必要地添加更多容器，因为我们不允许我们的应用程序扩展到其“最佳位置”。让我们更详细地探讨一下这个重要"sweet spot"的 a 概念。

## 正确调整应用程序大小

每个应用程序都有一个特定的点，它无法再占用流量。超过此点可能会增加处理时间，甚至在推送到远远超过此点时会降低流量。这被称为应用程序的饱和点。为避免缩放问题，我们应该尝试在应用程序达到饱和点之前对其进行缩放。让我们称这一点为最佳位置。

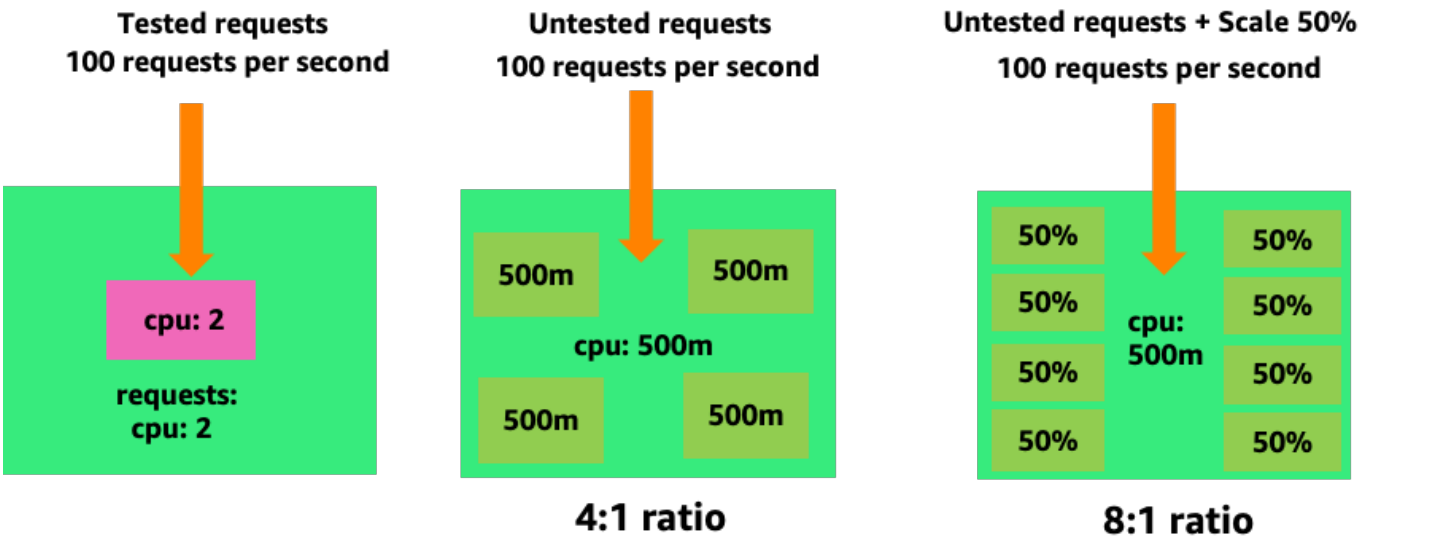



我们需要测试每个应用程序，以了解其最佳位置。由于每个应用程序都不同，因此这里没有通用的指导。在这次测试中，我们试图了解显示应用程序饱和点的最佳指标。通常，利用率指标被用来表示应用程序已饱和，但这很快就会导致扩展问题（我们将在后面的章节中详细探讨这个主题）。一旦我们有了这个“最佳位置”，我们就可以用它来有效地扩展我们的工作负载。

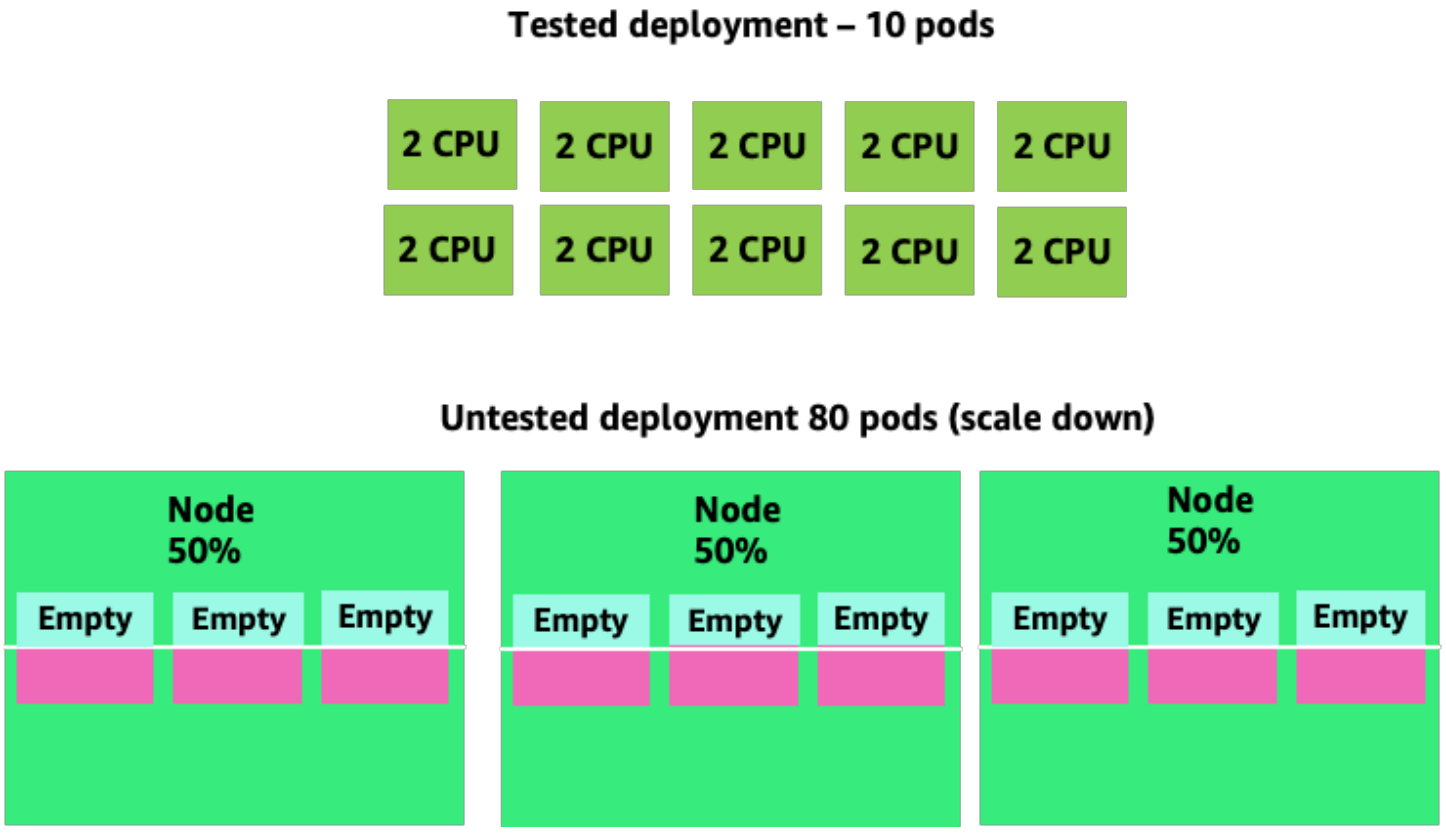
相反，如果我们在最佳位置之前就扩大规模并创建不必要的 pod，会发生什么？ 让我们在下一节中探讨这个问题。

## 吊舱蔓延

要了解创建不必要的 pod 会如何很快失控，让我们看一下左边的第一个示例。当每秒处理 100 个请求时，此容器的正确垂直缩放会占用大约两个 vCPU 的利用率。但是，如果我们通过将请求设置为半个内核来少调请求值，那么现在我们实际需要的每个 pod 需要 4 个 pod。更糟糕的是，如果我们的 HPA 设置为默认的50%CPU，则这些吊舱将缩放一半为空，从而产生 8:1 的比例。

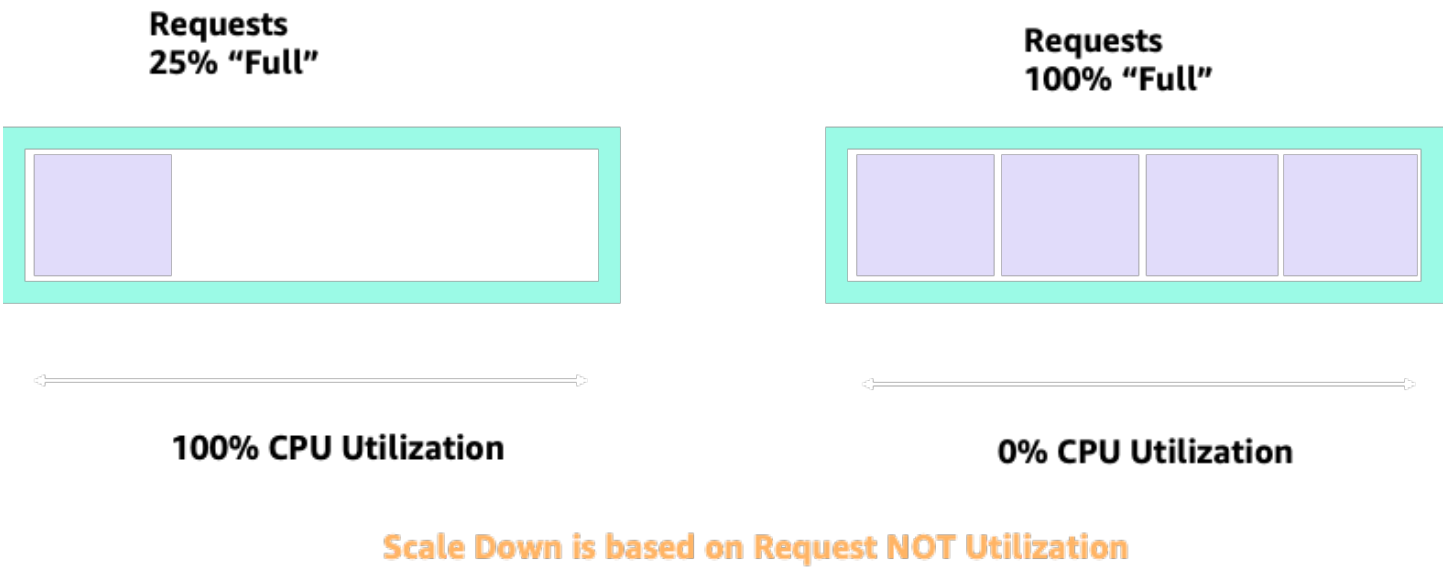



放大这个问题，我们可以很快看到这个问题是如何失控的。部署十个最佳位置设置不正确的 pod 可能会迅速上升到 80 个 pod 以及运行它们所需的额外基础架构。



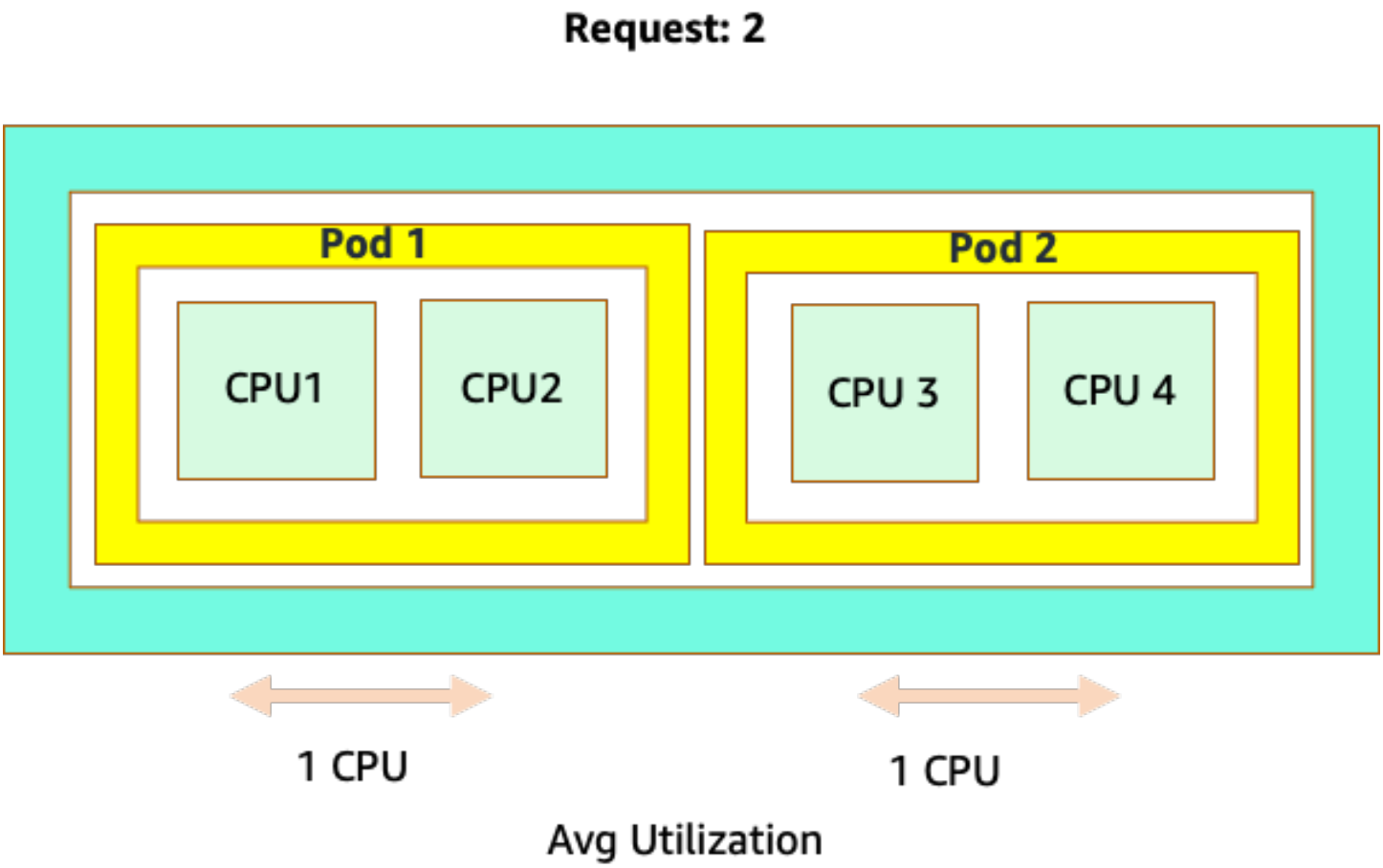

既然我们已经了解了不允许应用程序在其最佳位置运行的影响，那么让我们回到节点级别，问一下为什么 Kubernetes 调度器和 Linux CFS 之间的这种区别如此重要？

使用 HPA 向上和向下扩展时，我们可能会遇到这样一种情况：我们有足够的空间来分配更多 pod。这将是一个错误的决定，因为左边所示的节点已经达到了 100% 的 CPU 利用率。在一个不切实际但理论上可能的场景中，我们可能会遇到另一个极端，即我们的节点完全已满，但我们的 CPU 利用率为零。

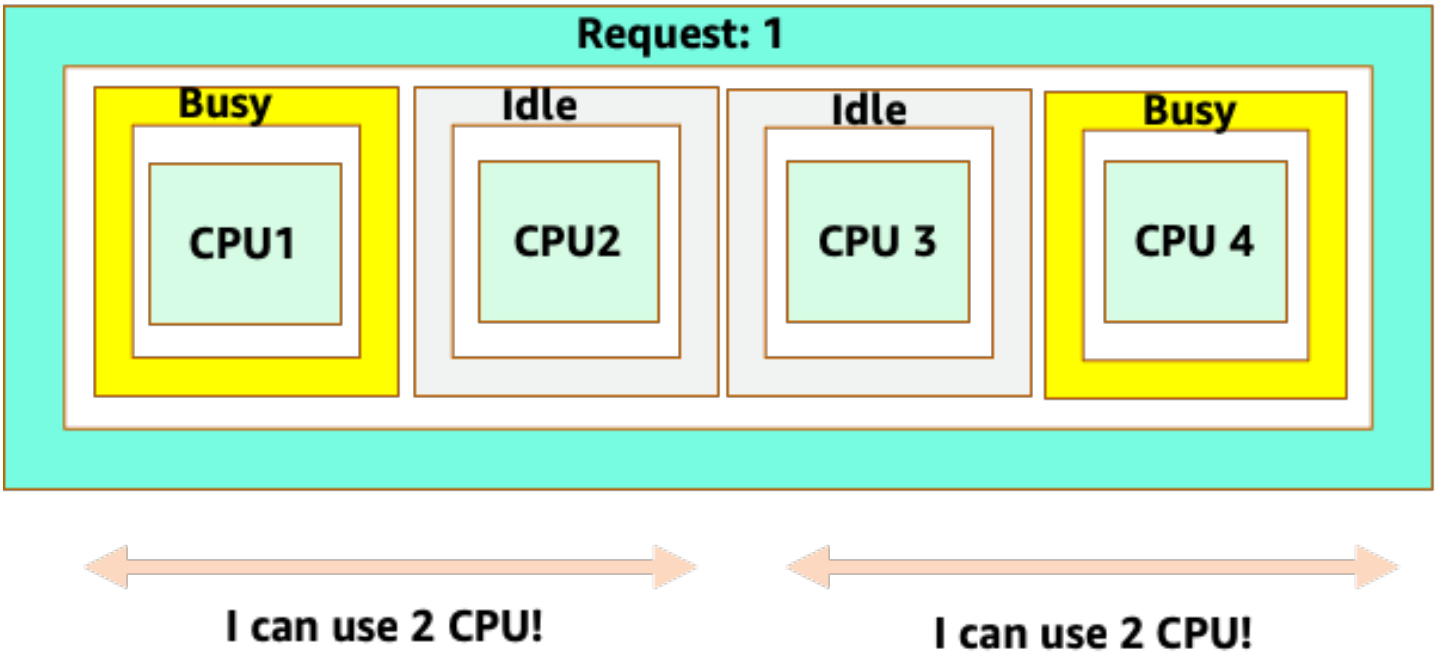



## 设置请求

将请求设置为该应用程序的 “最佳位置” 值很诱人，但这会导致效率低下，如下图所示。这里我们将请求值设置为 2 个 vCPU，但是这些 pod 的平均利用率大部分时间仅运行 1 个 CPU。这种设置会导致我们浪费 50% 的 CPU 周期，这是不可接受的。




这使我们想到了问题的复杂答案。容器利用率不能在真空中考虑；必须考虑节点上运行的其他应用程序。在以下示例中，本质上是爆发的容器与两个可能受内存限制的 CPU 利用率低的容器混合在一起。通过这种方式，我们允许容器在不对节点造成负担的情况下达到最佳位置。



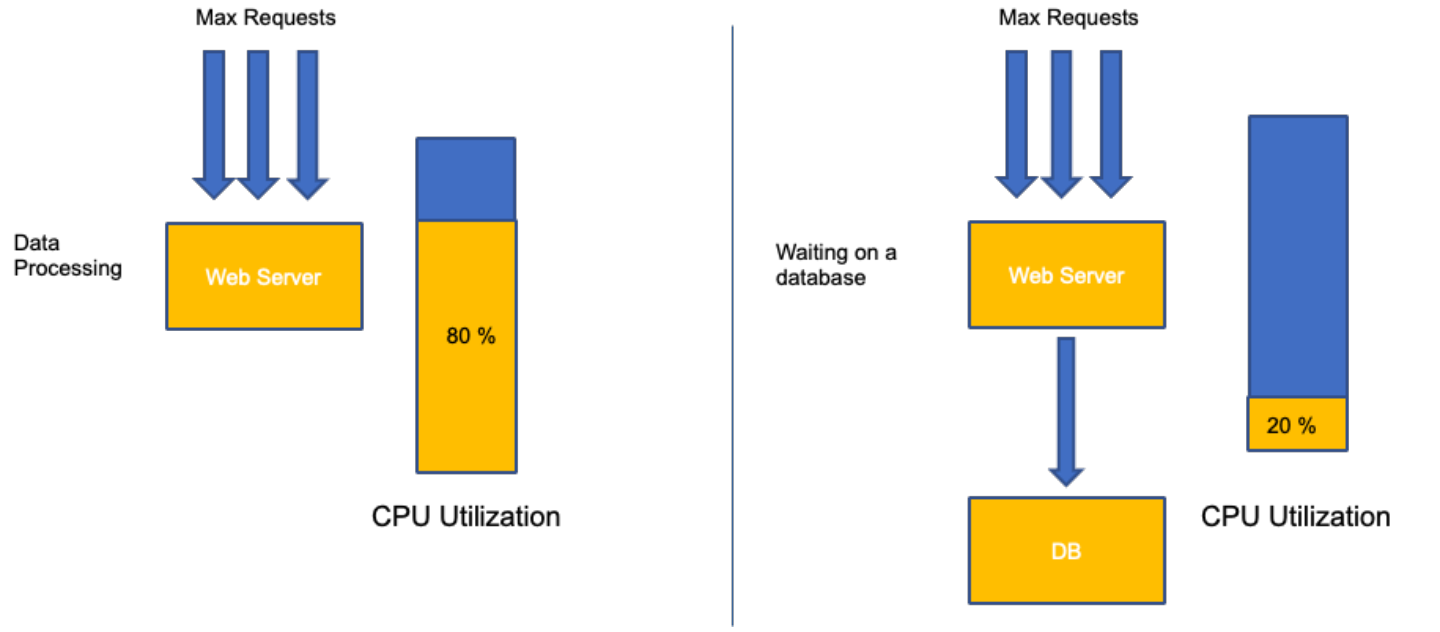

要从这一切中汲取的重要概念是，使用 Kubernetes 调度程序的内核概念来理解 Linux 容器性能可能会导致决策不佳，因为它们并不相关。

> **Note**
>
> Linux CFS 有其优点。对于 I/O 基于的工作负载尤其如此。但是，如果您的应用程序使用不带边车的完整内核，并且没有 I/O 要求，则CPU固定可以消除此过程中的大量复杂性，因此建议您注意这些警告。

## 利用率与饱和度

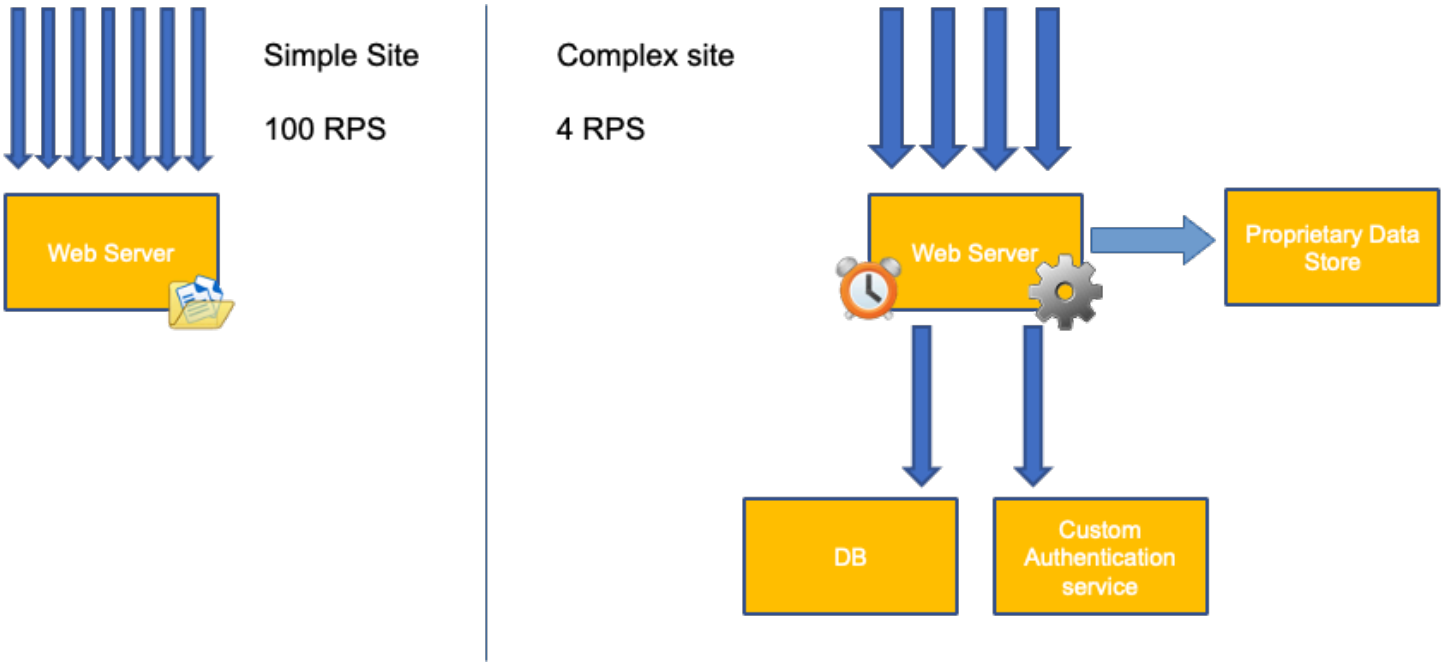
应用程序扩展中的一个常见错误是仅将 CPU 利用率作为扩展指标。在复杂的应用程序中，这几乎总是一个很差的指标，表明应用程序实际上已经充满了请求。在左边的示例中，我们看到我们所有的请求实际上都在发送 Web 服务器，因此 CPU 利用率在饱和度方面表现良好。

在现实世界的应用程序中，其中一些请求很可能会由数据库层或身份验证层等提供服务。在这种更常见的情况下，请注意 CPU 没有跟踪饱和度，因为请求是由其他实体提供服务的。在这种情况下，CPU 是衡量饱和度的一个非常差的指标。




在应用程序性能中使用错误的指标是 Kubernetes 中出现不必要和不可预测的扩展的首要原因。在为所使用的应用程序类型选择正确的饱和度指标时，必须格外小心。值得注意的是，没有可以给出“一刀切”的建议。根据所使用的语言和相关应用程序的类型，有各种各样的饱和度指标。

我们可能认为这个问题只出在 CPU 利用率上，但是其他常见指标（例如每秒请求数）也可能属于与上面讨论的完全相同的问题。请注意，请求也可能转到数据库层、身份验证层，而不是由我们的 Web 服务器直接提供服务，因此衡量 Web 服务器本身的真实饱和度是一个很差的指标。




不幸的是，在选择正确的饱和度指标时，没有简单的答案。以下是一些需要考虑的指导方针：

- 了解您的语言运行时——具有多个操作系统线程的语言的反应与单线程应用程序不同，因此对节点的影响也不同。
- 了解正确的垂直缩放比例——在缩放新 pod 之前，你想在应用程序的垂直比例中放多少缓冲区？
- 哪些指标能真正反映应用程序的饱和度-Kafka Producer 的饱和度指标将与复杂的 Web 应用程序大不相同。
- 节点上的所有其他应用程序如何相互影响-应用程序性能不是在真空中完成的，节点上的其他工作负载会产生重大影响。

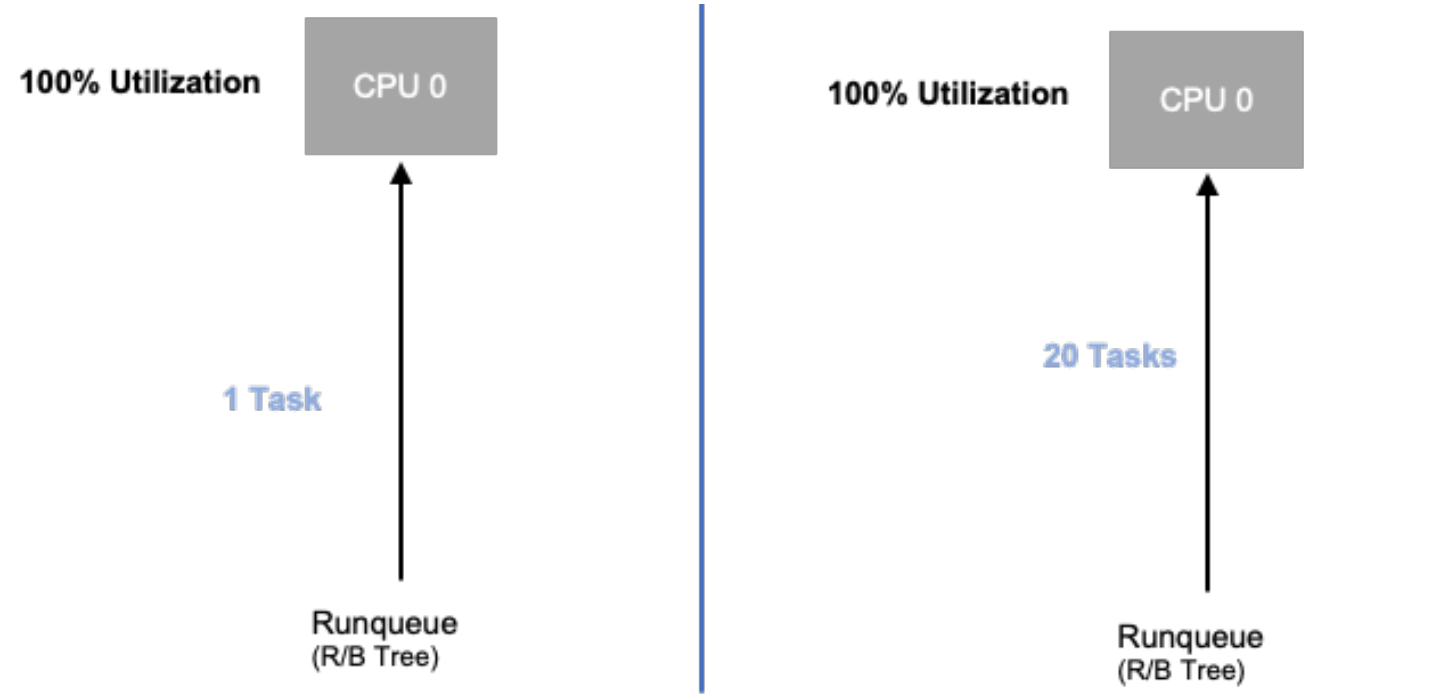
为了结束本节，很容易将上述内容视为过于复杂和不必要。我们经常会遇到问题，但我们没有意识到问题的真正本质，因为我们看错了指标。在下一节中，我们将探讨如何发生这种情况。

## 节点饱和度

现在我们已经探讨了应用程序饱和度，让我们从节点的角度来看同样的概念。让我们以两个 100% 利用率的 CPU 来了解利用率与饱和度之间的区别。

左边的 vCPU 利用率为 100%，但是没有其他任务等待在这个 vCPU 上运行，因此从纯粹的理论意义上讲，这非常高效。同时，在第二个示例中，我们有 20 个单线程应用程序等待由 vCPU 处理。现在，所有 20 个应用程序在等待轮到 vCPU 处理时都会遇到某种类型的延迟。换句话说，右边的 vCPU 已饱和。

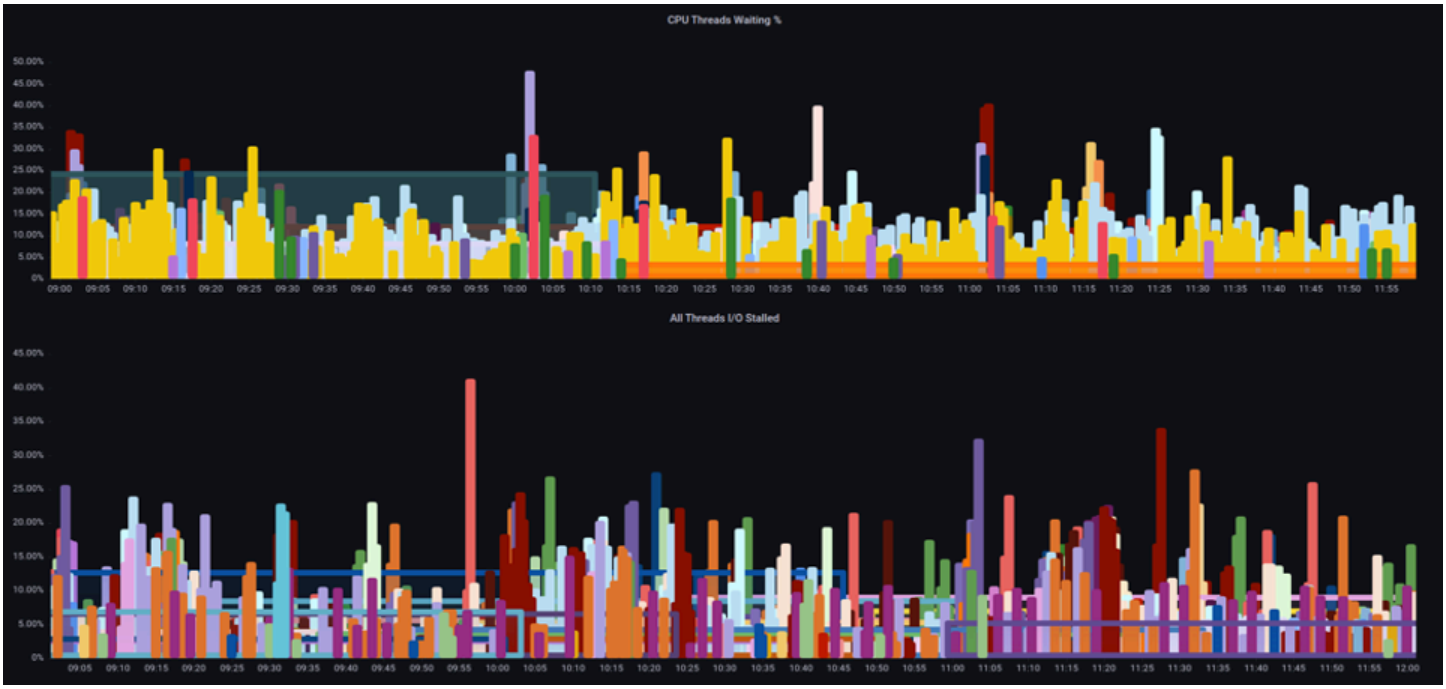
如果我们只看利用率，我们不仅不会看到这个问题，而且我们可能会将这种延迟归因于不相关的东西，例如网络，这会导致我们走上错误的道路。




在任何给定时间增加节点上运行的 Pod 总数时，重要的是要查看饱和度指标，而不仅仅是利用率指标，因为我们很容易忽略节点过饱和的事实。对于此任务，我们可以使用压力失速信息指标，如下图所示。

PromQL-停滞不前 I/O

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```




> **Note**
>
> 有关压力失速指标的更多信息，请参阅 https://facebookmicrosites.github.io/psi/docs/overview *

通过这些指标，我们可以判断线程是否在 CPU 上等待，或者即使盒子上的每个线程都停滞不前，等待内存或 I/O等资源。例如，我们可以看到实例上的每个线程在 1 分钟内停滞等待的百分比。 I/O

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```

使用这个指标，我们可以在上面的图表中看到，盒子上的每个线程在最高水位等候的时间中有45%的时间 I/O 处于停滞状态，这意味着我们在那一分钟内浪费了所有CPU周期。了解这种情况的发生可以帮助我们节省大量的 vCPU 时间，从而提高扩展效率。

## HPA V2

建议使用 HPA API 的 autoscaling/v2 版本。在某些边缘情况下，旧版本的 HPA API 可能会卡住扩展。它还被限制为 pod 在每个扩展步骤中只能翻一番，这给需要快速扩展的小型部署带来了问题。

Autoscaling/v2 允许我们更灵活地包括多个标准来进行扩展，并且在使用自定义和外部指标（非 K8s 指标）时为我们提供了极大的灵活性。

例如，我们可以按三个值中最高的值进行缩放（见下文）。如果所有 pod 的平均利用率超过 50%，如果自定义指标每秒入口的数据包平均超过 1,000 个，或者入口对象超过每秒 1 万个请求，我们就会进行扩展。

> **Note**
>
> 这只是为了展示自动缩放 API 的灵活性，我们建议不要使用在生产中难以排除故障的过于复杂的规则。

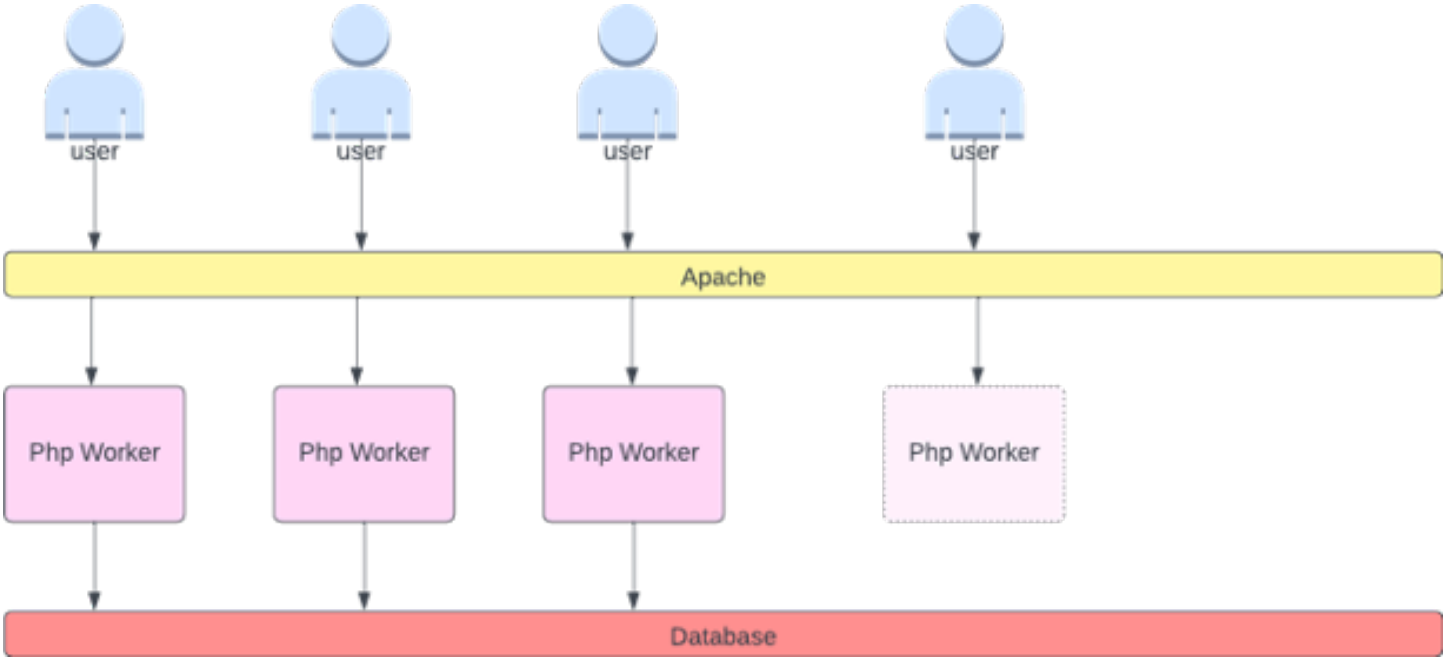
```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Pods
    pods:
      metric:
        name: packets-per-second
      target:
        type: AverageValue
        averageValue: 1k
  - type: Object
    object:
      metric:
        name: requests-per-second
      describedObject:
        apiVersion: networking.k8s.io/v1
        kind: Ingress
```

```
        name: main-route
      target:
        type: Value
        value: 10k
```

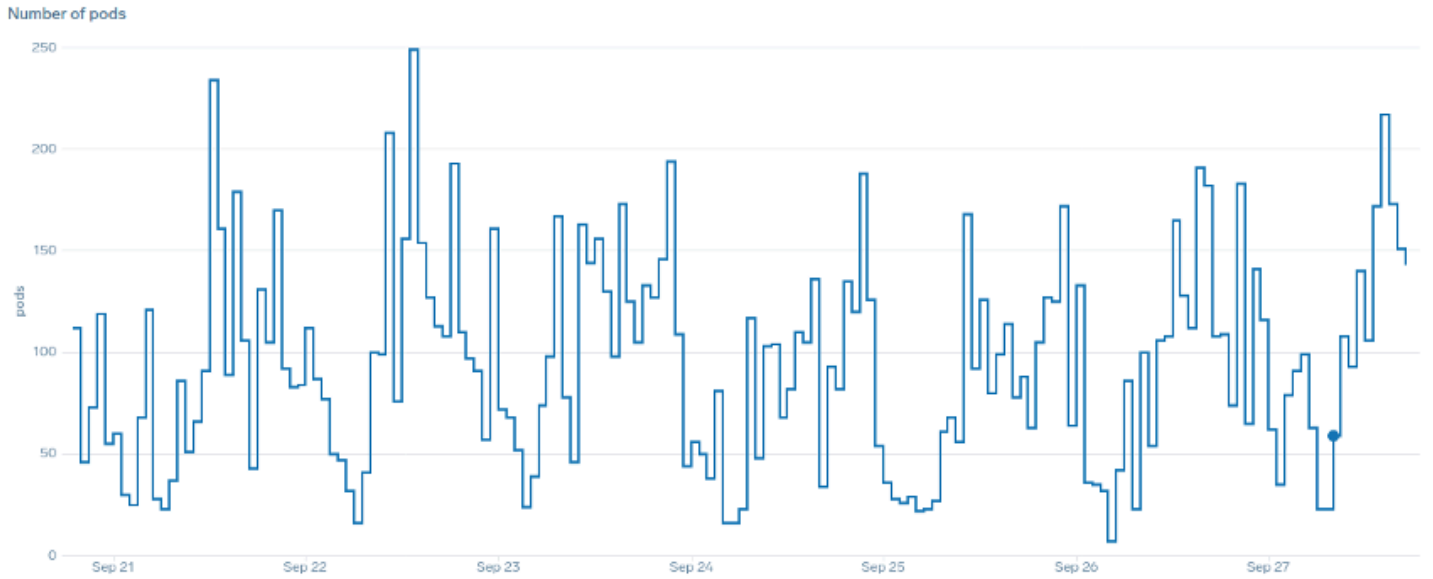
但是，我们了解了将此类指标用于复杂的 Web 应用程序的危险。在这种情况下，使用能够准确反映应用程序饱和度与利用率相比的自定义或外部指标可以更好地为我们服务。HPav2 通过能够根据任何指标进行扩展来实现这一点，但是我们仍然需要找到该指标并将其导出到 Kubernetes 以供使用。

例如，我们可以查看 Apache 中的活动线程队列数。这通常会创建一个 “更平滑” 的缩放配置文件（稍后会详细介绍该术语）。如果线程处于活动状态，则无论该线程是在数据库层等待还是在本地处理请求，如果所有应用程序线程都在使用中，则很好地表明应用程序已饱和。

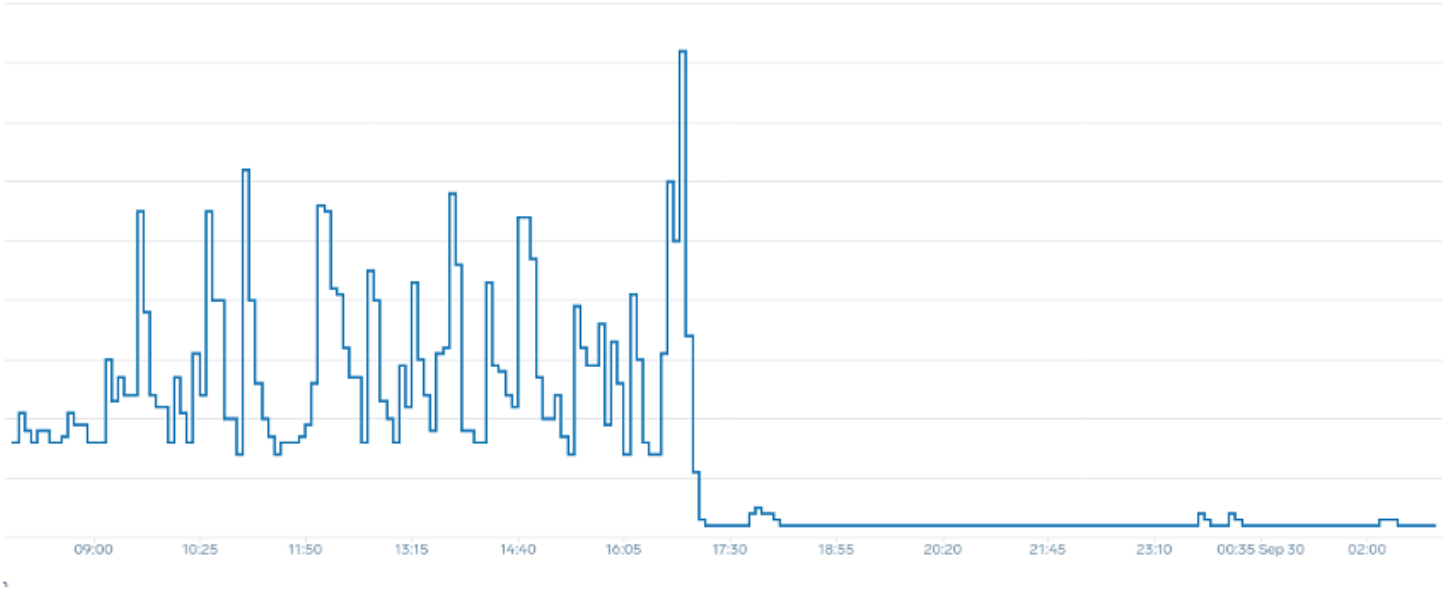
我们可以使用这种线程耗尽作为信号，创建一个线程池完全可用的新 pod。这也使我们能够控制在流量繁忙时要在应用程序中吸收多大的缓冲区。例如，如果我们的线程池总数为 10，则按使用的 4 个线程而不是使用的 8 个线程进行扩展将对我们在扩展应用程序时可用的缓冲区产生重大影响。对于需要在重负载下快速扩展的应用程序来说，设置为 4 是有意义的，如果我们有足够的时间进行扩展，因为请求数量的增长缓慢而不是随着时间的推移而急剧增加，那么如果我们有足够的时间进行扩展，那么设置为 8 会更有效地利用我们的资源。




在缩放方面，“平滑” 一词是什么意思？ 请注意下图，其中我们使用 CPU 作为指标。此部署中的 pod 将在短时间内激增，从 50 个 pod 一直到 250 个 pod，然后立即再次缩小。这是效率极低的扩展是集群流失的主要原因。




请注意，在我们更改为反映应用程序正确最佳位置的指标（图表的中间部分）之后，我们如何能够平稳地进行扩展。现在，我们的缩放效率很高，我们的 pod 可以根据我们通过调整请求设置提供的余量进行完全扩展。现在，一小群吊舱正在做以前数百个吊舱所做的工作。现实世界的数据表明，这是 Kubernetes 集群可扩展性的首要因素。

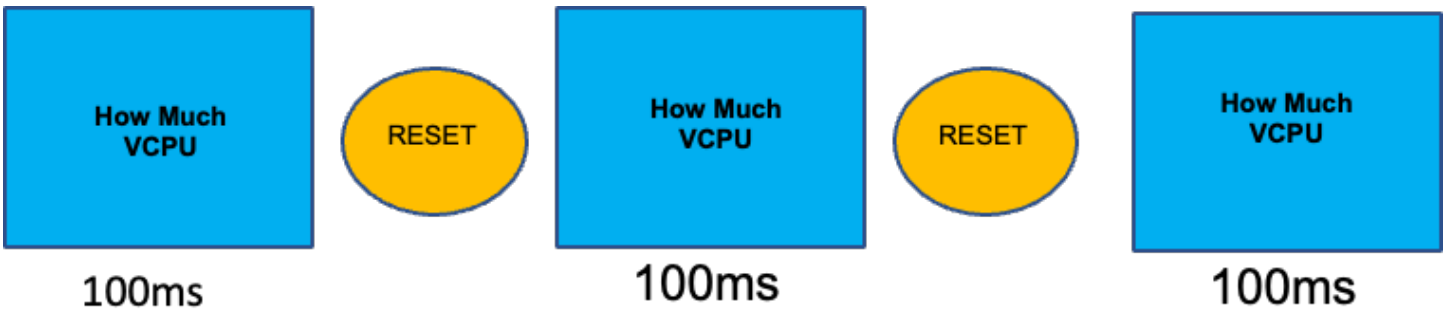



关键要点是 CPU 利用率只是应用程序和节点性能的一个维度。使用 CPU 利用率作为节点和应用程序的唯一运行状况指标会带来扩展、性能和成本方面的问题，这些都是紧密联系的概念。应用程序和节点的性能越高，需要扩展的次数就越少，这反过来又会降低成本。

通过查找和使用正确的饱和度指标来扩展您的特定应用程序，您还可以监控该应用程序的真正瓶颈并发出警报。如果跳过这一关键步骤，将很难甚至不可能理解有关性能问题的报告。

# 设置 CPU 限制

为了完善本节关于被误解的话题，我们将介绍 CPU 限制。简而言之，限制是与容器关联的元数据，该容器具有每 100 毫秒重置一次的计数器。这有助于 Linux 跟踪特定容器在 100 毫秒的时间段内在节点范围内使用了多少 CPU 资源。







设置限制时的一个常见错误是假设应用程序是单线程的，并且只能在其 “`分配的” vCPU上运行。在上一节中，我们了解到 CFS 不分配内核，实际上，运行大型线程池的容器将在盒子上所有可用的 vCPU 上进行调度。

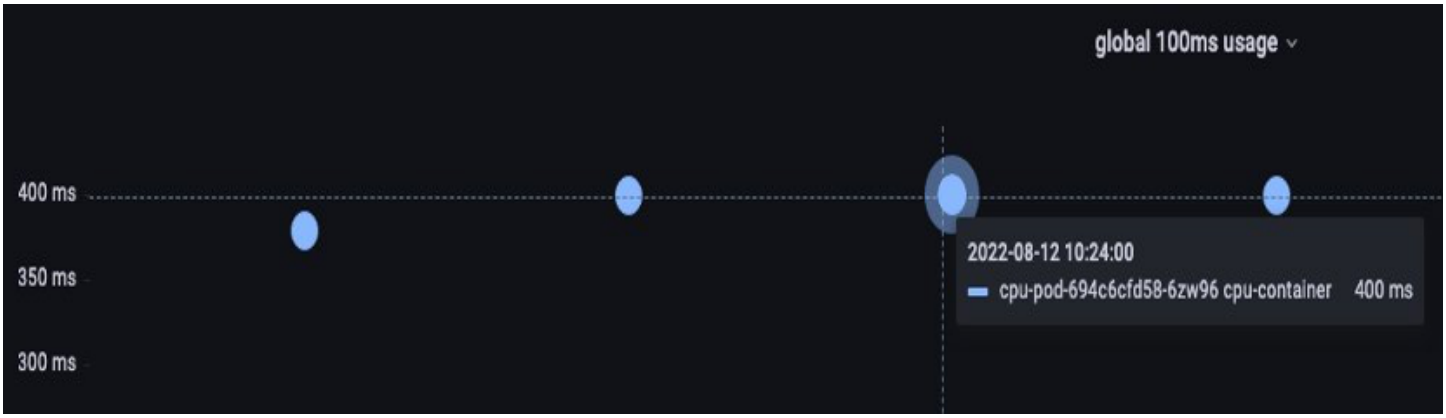
如果 64 个操作系统线程在 64 个可用内核上运行（从 Linux 节点的角度来看），则在将所有 64 个内核上的运行时间相加后，我们将在将 100 毫秒内的 CPU 使用时间总账单计算得相当大。由于这可能只发生在垃圾收集过程中，因此很容易错过这样的事情。这就是为什么在尝试设置限制之前，有必要使用指标来确保随着时间的推移我们有正确的使用情况。

幸运的是，我们可以准确地看到应用程序中所有线程使用了多少 vCPU。我们将使用该指标来`container_cpu_usage_seconds_total`实现此目的。

由于限制逻辑每 100 毫秒发生一次，并且该指标是每秒的指标，因此我们将 PromQL 与这个 100 毫秒周期相匹配。如果您想深入了解这个 PromQL 语句的工作原理，请参阅以下博客。

PromQL 查询：

```
topk(3, max by (pod, container)(rate(container_cpu_usage_seconds_total{image!="",
 instance="$instance"}[$__rate_interval]))) / 10
```




一旦我们觉得自己有正确的价值，我们就可以限制生产。然后，有必要查看我们的应用程序是否由于意想不到的事情而受到限制。我们可以通过看一下来做到这一点 `container_cpu_throttled_seconds_total`

```
topk(3, max by (pod, container)(rate(container_cpu_cfs_throttled_seconds_total{image!
=``""``, instance=``"$instance"``}[$__rate_interval]))) / 10
```




## 内存

内存分配是另一个很容易将 Kubernetes 调度行为与 Linux cGroup 行为混为一谈的例子。这是一个更加细致入微的话题，因为 cGroup v2 在 Linux 中处理内存的方式发生了重大变化，而 Kubernetes 也更改了语法以反映这一点；阅读此博客了解更多详情。

与 CPU 请求不同，在调度过程完成后，内存请求将处于未使用状态。这是因为我们无法像使用 CPU 那样在 cGroup v1 中压缩内存。这只给我们留下了内存限制，这些限制旨在通过完全终止 pod 来保护内存泄漏。这是一个全有要么全有要么全无的风格主张，但是我们现在有了解决这个问题的新方法。

首先，重要的是要明白，为容器设置正确的内存量并不像看起来那样简单。Linux 中的文件系统将使用内存作为缓存来提高性能。这种缓存会随着时间的推移而增长，很难知道有多少内存可以用来存放缓

存，但可以在不对应用程序性能产生重大影响的情况下进行回收。这通常会导致对内存使用情况的误解。

具有 “压缩” 内存的能力是 cGroup v2 背后的主要驱动力之一。有关为何需要 cGroup V2 的更多历史记录，请参阅 Chris Down 在 LISA21 上的演讲，他在演讲中介绍了为什么无法正确设置最小内存是促使他创建 cGroup v2 和压力失速指标的原因之一。

幸运的是，Kubernetes 现在有了 “和” 之下的概念。`memory.min` `memory.high` `requests.memory`这使我们可以选择主动释放缓存的内存供其他容器使用。一旦容器达到内存上限，内核就可以积极回收该容器的内存，最高可达设置为的值。`memory.min`因此，当节点承受内存压力时，我们可以获得更大的灵活性。

关键问题变成了，`memory.min`要设置为什么值？ 这就是内存压力失速指标发挥作用的地方。我们可以使用这些指标来检测容器级别的内存 “抖动”。然后，我们可以使用诸如 fbtax 之类的控制器`memory.min`通过查找内存抖动来检测正确的值，并将该`memory.min`值动态设置为该设置。

### Summary

总而言之，可以很容易地将以下概念混为一谈：

- 利用率和饱和度
- 使用 Kubernetes 调度器逻辑的 Linux 性能规则

必须格外小心，将这些概念分开。性能和规模在深层次上是相互关联的。不必要的缩放会造成性能问题，进而造成扩展问题。

## Kubernetes 上游 SLO

Amazon EKS 运行的代码与上游 Kubernetes 版本相同，并确保 EKS 集群在 Kubernetes 社区定义的 SLO 内运行。Kubernetes 可扩展性特别兴趣小组 (SIG) 定义了可扩展性目标，并通过 SLI 和 SLO 调查性能瓶颈。

SLI 是我们衡量系统的方式，例如指标或衡量标准，可用于确定系统运行的 “良好”，例如请求延迟或计数。SLO 定义了系统运行 “良好” 时的预期值，例如请求延迟保持在 3 秒以内。Kubernetes SLO 和 SLI 侧重于 Kubernetes 组件的性能，完全独立于注重 EKS 集群终端节点可用性的亚马逊 EKS 服务 SLA。

Kubernetes 具有许多功能，允许用户使用自定义插件或驱动程序扩展系统，例如 CSI 驱动程序、准入 webhook 和自动缩放器。这些扩展可能会以不同的方式极大地影响 Kubernetes 集群的性能，也就是

说，如果 webhook 目标不可用，则带有准入 webhook 的 webhook `failurePolicy=Ignore` 可能会增加 K8s API 请求的延迟。Kubernetes 可扩展性 SIG 使用 “你承诺，我们承诺” 的框架来定义可扩展性：

> 如果您承诺：-正确配置您的集群-“合理” 使用可扩展性功能-将集群中的负载保持在建议的限制之内
>
> 那么我们保证您的集群会扩展，即：-满足所有 SLO。

## Kubernetes SLO

Kubernetes SLO 并未考虑所有可能影响集群的插件和外部限制，例如工作节点扩展或准入 webhook。这些 SLO 侧重于 Kubernetes 组件，并确保 Kubernetes 的操作和资源在预期范围内运行。SLO 可帮助 Kubernetes 开发人员确保对 Kubernetes 代码的更改不会降低整个系统的性能。

Kuberntes 可扩展性 SIG 定义了以下官员。 SLO/SLIsAmazon EKS 团队定期在 EKS 集群上运行可扩展性测试， SLOs/SLIs 以监控随着更改和新版本发布而出现的性能下降。

| 目标 | 定义 | SLO |
| --- | --- | --- |
| API 请求延迟（变更） | 处理每对（资源、动词）的单个对象的变异 API 调用的延迟，以过去 5 分钟内的第 99 个百分位数来衡量 | 在默认的 Kubernetes 安装中，对于每对（资源、动词），不包括虚拟资源和聚合资源以及自定义资源定义，每个集群日的第 99 个百分位数 <= 1s |
| API 请求延迟（只读） | 处理每个（资源、范围）对的非流式只读 API 调用的延迟，以过去 5 分钟内的第 99 个百分位数来衡量 | 在默认 Kubernetes 安装中，对于每对（资源、范围），不包括虚拟和聚合资源以及自定义资源定义，每个集群日的第 99 个百分位数：(a) 如果 (b) <= 1s 否则为 <= 30sscope=resource （如果或）scope=namespace scope=cluster |
| Pod 启动延迟 | 可调度的无状态 pod 的启动延迟，不包括拉取镜像和运行 init 容器的时间，从 pod 创建时间 | 在默认的 Kubernetes 安装中，每个集群日的第 99 个百分位数 <= 5s |

| 目标 | 定义 | SLO |
| --- | --- | --- |
| | 戳到所有容器都报告为已启动并通过 watch 观察的时间来衡量，以过去 5 分钟内的第 99 个百分位数来衡量 | |

## API 请求延迟

`kube-apiserver`已`--request-timeout`定义为`1m0s`默认值，这意味着请求最多可以运行一分钟（60 秒），然后才会被超时和取消。为延迟定义的 SLO 按正在发出的请求类型进行细分，这些类型可以是变异的，也可以是只读的：

变异

Kubernetes 中的变异请求会对资源进行更改，例如创建、删除或更新。这些请求非常昂贵，因为在返回更新的对象之前，必须将这些更改写入 etcd 后端。Etcd 是一种分布式键值存储，用于所有 Kubernetes 集群数据。

这种延迟是以 Kubernetes 资源对（资源、动词）在 5 分钟内的第 99 个百分位来衡量的，例如，这将衡量 Create Pod 请求和更新节点请求的延迟。请求延迟必须为 <= 1 秒才能满足 SLO。

Read-only

Read-only 请求检索单个资源（例如 Get Pod X）或集合（例如 “从命名空间 X 获取所有 Pod”）。`kube-apiserver`维护对象的缓存，因此可以从缓存中返回请求的资源，或者可能需要先从 etcd 中检索它们。这些延迟也是以 5 分钟内的第 99 个百分位数来衡量的，但是只读请求可以有不同的范围。SLO 定义了两个不同的目标：

- 对于针对单个资源（即`kubectl get pod -n mynamespace my-controller-xxx`）发出的请求，请求延迟应保持 <= 1 秒。
- 对于对命名空间或集群（例如`kubectl get pods -A`）中的多个资源发出的请求，延迟应保持 <= 30 秒

对于不同的请求范围，SLO 具有不同的目标值，因为对 Kubernetes 资源列表发出的请求需要在 SLO 中返回请求中所有对象的详细信息。在大型集群或大型资源集合上，这可能会导致响应大小过大，可能需要一些时间才能返回。例如，在运行成千上万个 Pod 且每个 Pod 以 JSON 编码时大约 1 KiB

的集群中，返回集群中的所有 Pod 将包含 10MB 或更多。Kubernetes 客户端可以通过检索大量资源 APIListChunking 来帮助缩小响应大小。

### Pod 启动延迟

这个 SLO 主要关注从 Pod 创建 Pod 到该 Pod 中的容器实际开始执行所花费的时间。为了衡量这一点，计算了与 Pod 上记录的创建时间戳以及该 Pod 上的 WAT CH 报告容器已启动时的差异（不包括容器镜像拉取和初始化容器执行时间）。为了满足 SLO，此 Pod 启动延迟中每个集群日的第 99 个百分位数必须保持 <=5 秒。

请注意，此 SLO 假设此集群中已经存在工作节点，处于可以调度 Pod 的就绪状态。此 SLO 不考虑图像拉取或初始化容器执行，还将测试限制在不利用持久存储插件的 “无状态 pod” 上。

## Kubernetes SLI 指标

Kubernetes 还通过向随时间跟踪这些 SLI 的 Kubernetes 组件添加 Prometheus 指标来改善围绕 SL I 的可观察性。使用 Prometheus 查询语言 (PromQL)，我们可以构建查询，在 Prom etheus 或 Grafana 仪表板等工具中显示一段时间内的 SLI 性能，以下是上述 SLI 的一些示例。

### API 服务器请求延迟

| 指标 | 定义 |
| --- | --- |
| apiserver_request_sli_duration_seconds | 每个动词、组、版本、资源、子资源、范围和组件的响应延迟分布（不包括 webhook 持续时间以及优先级和公平性队列等待时间），以秒为单位。 |
| apiserver_request_持续时间_秒 | 每个动词、试运行值、组、版本、资源、子资源、范围和组件的响应延迟分布（以秒为单位）。 |

> **Note**
>
> 该`apiserver_request_sli_duration_seconds`指标从 Kubernetes 1.27 开始可用。

您可以使用这些指标来调查 API 服务器的响应时间以及 Kubernetes 组件或其他组件中是否存在瓶颈。plugins/components以下查询基于社区 SLO 控制面板。

API 请求延迟 SLI（变异）-此时间不包括 webhook 的执行或队列中的等待时间。
```
histogram_quantile(0.99,
sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"CREATE|
DELETE|PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}
[5m])) by (resource, subresource, verb, scope, le)) > 0
```

API 请求延迟总计（变更）-这是请求在 API 服务器上花费的总时间，此时间可能比 SLI 时间长，因为它包括 webhook 执行以及 API 优先级和公平性等待时间。
```
histogram_quantile(0.99,
sum(rate(apiserver_request_duration_seconds_bucket{verb=~"CREATE|DELETE|
PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}[5m])) by
(resource, subresource, verb, scope, le)) > 0
```

在这些查询中，我们排除了不会立即返回的流式传输 API 请求，例如`kubectl port-forward`或`kubectl exec`请求 (`subresource!~"proxy|attach|log|exec|portforward"`)，并且我们只筛选修改对象的 Kubernetes 动词 ()。`verb=~"CREATE|DELETE|PATCH|POST|PUT"`然后，我们将计算过去 5 分钟内延迟的第 99 个百分位数。

我们可以对只读 API 请求使用类似的查询，我们只需修改要筛选的动词以包含只读操作`LIST`和`GET`。根据请求的范围，也存在不同的 SLO 阈值，即获取单个资源或列出多个资源。

API 请求延迟 SLI（只读）-此时间不包括 webhook 的执行或队列中的等待时间。对于单个资源（范围=资源，阈值=1s）
```
histogram_quantile(0.99,
sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"GET",
scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))
```

对于同一命名空间中的资源集合（scope=namespace，threshold=5s）
```
histogram_quantile(0.99,
sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST",
scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))
```

对于整个集群中的资源集合（scope=cluster，threshold=30s）
```
histogram_quantile(0.99,
sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST",
scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))
```

API 请求延迟总计（只读）-这是请求在 API 服务器上花费的总时间，此时间可能比 SLI 时间长，因为它包括 webhook 的执行和等待时间。对于单个资源（范围=资源，阈值=1s）
```
histogram_quantile(0.99,
sum(rate(apiserver_request_duration_seconds_bucket{verb=~"GET",
scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))
```

对于同一命名空间中的资源集合
（scope=namespace，threshold=5s）`histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))`

对于整个集群中的资源集合（scope=cluster，threshold=30s）`histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))`

SLI 指标通过排除请求在 API 优先级和公平性队列中等待所花费的时间、通过准入 webhook 或其他 Kubernetes 扩展程序运行所花费的时间，从而深入了解 Kubernetes 组件的性能。总指标提供了更全面的视图，因为它反映了您的应用程序等待 API 服务器响应的时间。比较这些指标可以深入了解请求处理延迟在哪里引入的。

## Pod 启动延迟

| 指标 | 定义 |
| --- | --- |
| kubelet_pod_start_sli_sli_duration_seconds | 启动 Pod 的持续时间（以秒为单位），不包括拉取图像和运行 init 容器的时间，从 pod 创建时间戳到其所有容器都报告为已启动并通过 watch 进行观察 |
| kubelet_pod_start_duration_seconds | 从 kubelet 第一次看到 pod 到 pod 开始运行的持续时间（以秒为单位）。这不包括调度 Pod 或扩展工作节点容量的时间。 |

> **Note**
>
> `kubelet_pod_start_sli_duration_seconds`从 Kubernetes 1.27 开始可用。

与上面的查询类似，你可以使用这些指标来深入了解与 Kubelet 操作相比，节点缩放、图像拉取和初始化容器延迟 pod 启动的时间有多长。

Pod 启动延迟 SLI- 这是从创建 Pod 到应用程序容器报告运行的时间。
这包括工作节点容量可用和调度 pod 所花费的时间，但这不包括拉取映

像或初始化容器运行所花费的时间。 `histogram_quantile(0.99, sum(rate(kubelet_pod_start_sli_duration_seconds_bucket[5m])) by (le))`

Pod 启动延迟总计- 这是 kubelet 首次启动 pod 所花费的时间。这是从 kubelet 通过 WATCH 收到 pod 的时间开始计算的，其中不包括工作节点扩展或调度的时间。这包括拉取映像和初始化容器以供运行的时间。 `histogram_quantile(0.99, sum(rate(kubelet_pod_start_duration_seconds_bucket[5m])) by (le))`

## 集群上的 SLO

如果您正在从 EKS 集群中的 Kubernetes 资源中收集 Prometheus 指标，则可以更深入地了解 Kubernetes 控制平面组件的性能。

性能测试存储库包含 Grafana 仪表板，用于显示测试期间集群的延迟和关键性能指标。性能测试配置利用了 kube-p rometheus-stack，这是一个配置为收集 Kub ernetes 指标的开源项目，但你也可以使用亚马逊托管 Prometheus 和亚马逊托管 Grafana。

如果您使用的是`kube-prometheus-stack`或类似的 Prometheus 解决方案，则可以安装相同的控制面板来实时观察集群上的 SLO。

1. 您首先需要安装控制面板中使用的 Prometheus 规则。`kubectl apply -f prometheus-rules.yaml`你可以在这里下载规则的副本：https://github.com/kubernetes/perf-tests/blob/master/clusterloader2/pkg/prometheus/manifests/prometheus-rules.yaml
   a. 请务必检查文件中的命名空间是否与您的环境相匹配
   b. 如果您使用的是，请验证标签是否与 `prometheus.prometheusSpec.ruleSelector` helm 值匹配 `kube-prometheus-stack`
2. 然后，您可以在 Grafana 中安装控制面板。这里提供了 json 仪表板和生成它们的 python 脚本：https://github.com/kubernetes/perf-tests/tree/master/clusterloader2/pkg/prometheus/manifests/dashboards
   a. `slo.json`控制面板显示集群与 Kubernetes SLO 相关的性能

考虑一下，SLO 侧重于集群中 Kubernetes 组件的性能，但是您可以查看其他指标，这些指标可以为集群提供不同的视角或见解。像 Kubernetes 这样的社区项目Kube-state-metrics可以帮助你快速分析集群中的趋势。Kubernetes 社区中大多数常见的插件和驱动程序也会发出 Prometheus 指标，允许你调查诸如自动扩缩程序或自定义调度器之类的内容。

《可观察性最佳实践指南》提供了其他 Kubernetes 指标的示例，您可以使用这些指标来获得进一步的见解。

# 已知限制和 Service Quotas

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

Amazon EKS 可用于各种工作负载，可以与各种 AWS 服务进行交互，而且我们已经看到客户工作负载会遇到类似的 AWS 服务配额和其他阻碍可扩展性的问题。

您的 AWS 账户具有默认配额（您的团队可以申请的每个 AWS 资源数量的上限）。每个 AWS 服务都定义自己的配额，配额通常是特定于区域的。您可以请求增加某些配额（软限制），而其他配额则无法提高（硬限制）。在设计应用程序时，您应该考虑这些值。请考虑定期查看这些服务限制，并在应用程序设计期间将其纳入其中。

您可以在 AWS S ervice Quotas 控制台或使用 AWS CLI 查看账户中的使用情况并提出增加配额的请求。请参阅相应的 AWS 服务的 AWS 文档，了解有关服务配额的更多详细信息以及有关增加配额的任何其他限制或通知。

> **Note**
>
> Amazon EKS Ser vice Quotas 列出了服务配额，并提供了请求增加配额的链接（如果有）。

## 其他 AWS Service Quotas

我们已经看到 EKS 客户受到下面列出的其他 AWS 服务配额的影响。其中一些可能仅适用于特定的用例或配置，但是您可以考虑您的解决方案在扩展时是否会遇到任何这些用例或配置。配额是按服务组织的，每个配额都有一个格式为 ID， L-XXXXXXXX 你可以用来在 AWS Service Quotas 控制台中查找

| 服务 | 配额 (L-xxxxx) | Impact | 身份证 (L-xxxxx) | 默认 |
|---|---|---|---|---|
| IAM | 每个账户的角色数： | 可以限制账户中集群或 IRSA 角色的数量。 | L-FE177D64 | 1000 |
| IAM | OpenId 按账户连接提供商 | 可以限制每个账户的集群数 | L-858F3967 | 100 |

| 服务 | 配额 (L-xxxxx) | Impact | 身份证 (L-xxxxx) | 默认 |
| --- | --- | --- | --- | --- |
| | | 量，IRSA 使用 OpenID Connect | | |
| IAM | 角色信任策略长度 | 可以限制 IRSA 的 IAM 角色关联的集群数量 | L-C07B4B0D | 2,048 |
| VPC | 每个网络接口的安全组数 | 可能会限制集群对网络的控制或连接 | L-2AFB9258 | 5 |
| VPC | 每个 VPC 的 IPv4 CIDR 块数 | 可以限制 EKS 工作节点的数量 | L-83CA0A9D | 5 |
| VPC | 每个路由表的路由数 | 可能会限制集群对网络的控制或连接 | L-93826ACB | 50 |
| VPC | 每个 VPC 的活动 VPC 对等连接 | 可能会限制集群对网络的控制或连接 | L-7E9ECCDB | 50 |
| VPC | 每个安全组的入站或出站规则。 | 可能会限制集群对网络的控制或连接，EKS 中的某些控制器会创建新规则 | L-0EA8095F | 50 |
| VPC | 每个区域的 VPC 数 | 可以限制每个账户的集群数量，也可以限制集群的网络控制或连接 | L-F678F1CE | 5 |

| 服务 | 配额 (L-xxxxx) | Impact | 身份证 (L-xxxxx) | 默认 |
| --- | --- | --- | --- | --- |
| VPC | 每个区域的互联网网关数 | 可以限制每个账户的集群数量，也可以限制集群的网络控制或连接 | L-A4707A72 | 5 |
| VPC | 每个区域的网络接口数 | 可以限制 EKS 工作节点的数量，也可以限制 Impact EKS 控制平面 scaling/update 活动的数量。 | L-DF5E4CA3 | 5000 |
| VPC | 网络地址使用情况 | 可以限制每个账户的集群数量，也可以限制集群的网络控制或连接 | L-BB24F6E5 | 64,000 |
| VPC | 对等网络地址的使用情况 | 可以限制每个账户的集群数量，也可以限制集群的网络控制或连接 | L-CD17FD4B | 128,000 |
| ELB | 每个网络负载均衡器的侦听器数 | 可以限制对集群流量入口的控制。 | L-57A373D6 | 50 |
| ELB | 每个区域的目标组数 | 可以限制对集群流量入口的控制。 | L-B22855CB | 3000 |

| 服务 | 配额 (L-xxxxx) | Impact | 身份证 (L-xxxxx) | 默认 |
| --- | --- | --- | --- | --- |
| ELB | 每个 Application Load Balancer 的目标数 | 可以限制对集群流量入口的控制。 | L-7E6692B2 | 1000 |
| ELB | 每个网络负载均衡器的目标数 | 可以限制对集群流量入口的控制。 | L-EEF1AD04 | 3000 |
| ELB | 每个 Network Load Balancer 每个可用区的目标数 | 可以限制对集群流量入口的控制。 | L-B211E961 | 500 |
| ELB | 每个区域每个目标组的目标数 | 可以限制对集群流量入口的控制。 | L-A0D0B863 | 1000 |
| ELB | 每个区域的应用程序负载均衡器数 | 可以限制对集群流量入口的控制。 | L-53DA6B97 | 50 |
| ELB | 每个区域的经典负载均衡器数 | 可以限制对集群流量入口的控制。 | L-E9E9831D | 20 |
| ELB | 每个区域的网络负载均衡器数 | 可以限制对集群流量入口的控制。 | L-69A177A2 | 50 |
| EC2 | 正在运行的 On-Demand 标准（A、C、D、H、I、M、R、T、Z）实例（作为最大 vCPU 数量） | 可以限制 EKS 工作节点的数量 | L-1216C47A | 5 |

| 服务 | 配额 (L-xxxxx) | Impact | 身份证 (L-xxxxx) | 默认 |
| --- | --- | --- | --- | --- |
| EC2 | 所有标准（A、C、D、H、I、M、R、T、Z）竞价型实例请求（作为最大 vCPU 数量） | 可以限制 EKS 工作节点的数量 | L-34B43A08 | 5 |
| EC2 | EC2-VPC 弹性 IP | 可以限制 NAT GW（从而限制 VPC）的数量，这可能会限制区域中的集群数量 | L-0263D0A3 | 5 |
| EBS | 每个区域的快照数 | 可以限制有状态工作负载的备份策略 | L-309BACF6 | 100000 |
| EBS | 通用型 SSD（gp3）卷存储（单位 TiB） | 可以限制 EKS 工作节点的数量或 PersistentVolume 存储空间 | L-7A658B76 | 50 |
| EBS | 通用型 SSD（gp2）卷存储（单位 TiB） | 可以限制 EKS 工作节点的数量或 PersistentVolume 存储空间 | L-D18FCD1D | 50 |
| ECR | 已注册的存储库 | 可以限制集群中的工作负载数量 | L-CFEB8E8D | 100000 |
| ECR | 每个存储库的镜像数 | 可以限制集群中的工作负载数量 | L-03A36CE1 | 20000 |

| 服务 | 配额 (L-xxxxx) | Impact | 身份证 (L-xxxxx) | 默认 |
| --- | --- | --- | --- | --- |
| SecretsManager | 每个区域的秘密 | 可以限制集群中的工作负载数量 | L-2F66C23C | 500,000 |

## AWS 请求限制

AWS 服务还实施请求限制，以确保它们保持高性能并可供所有客户使用。与 Service Quotas 类似，每个 AWS 服务都有自己的请求限制阈值。如果您的工作负载需要快速发出大量 API 调用，或者您注意到应用程序中存在请求限制错误，请考虑查看相应的 AWS 服务文档。

在大型集群中或集群大幅扩展时，围绕配置 EC2 网络接口或 IP 地址的 EC2 API 请求可能会遇到请求限制。下表显示了我们看到的客户遇到请求限制的一些 API 操作。您可以在关于速率限制的 EC2 文档中查看 EC2 速率限制默认值以及请求提高速率限制的步骤。

| 变异动作 | Read-only 行动 |
| --- | --- |
| AssignPrivateIpAddresses | DescribeDhcpOptions |
| AttachNetworkInterface | DescribeInstances |
| CreateNetworkInterface | DescribeNetworkInterfaces |
| DeleteNetworkInterface | DescribeSecurityGroups |
| DeleteTags | DescribeTags |
| DetachNetworkInterface | DescribeVpcs |
| ModifyNetworkInterfaceAttribute | DescribeVolumes |
| UnassignPrivateIpAddresses | |

## 其他已知限制

- Route 53 对 Route 53 API 的速率限制也相当低，为每秒 5 个请求。如果您有大量域名需要使用诸如外部 DNS 之类的项目进行更新，则更新域名时可能会出现速率限制和延迟。

- 某些 Nitro 实例类型的卷连接限制为 28 个，由 Amazon EBS 卷、网络接口和 NVMe 实例存储卷共享。如果您的工作负载正在安装大量 EBS 卷，则使用这些实例类型可以实现的 pod 密度可能会受到限制
- 每个 Ec2 实例可跟踪的连接数有上限。如果您的工作负载正在处理大量连接，则可能会出现通信失败或错误，因为已达到该最大值。您可以使用`conntrack_allowance_available`和`conntrack_allowance_exceeded`网络性能指标来监控 EKS 工作节点上跟踪的连接数。
- 在 EKS 环境中，根据上游指南，etcd 存储限制为 8 GiB。请监控指标`apiserver_storage_size_bytes`以跟踪 etcd db 大小。您可以参考警报规则`etcdBackendQuotaLowSpace`和`etcdExcessiveDatabaseGrowth`并设置此监控。

# 群集升级的最佳实践

> **Tip**
>
> 通过 Amazon EKS 研讨会@@ 探索最佳实践。

本指南向集群管理员展示了如何规划和执行他们的 Amazon EKS 升级策略。它还描述了如何升级自管节点、托管节点组、Karpenter 节点和 Fargate 节点。它不包括关于 EKS Anywhere、自我管理的 Kubernetes、AWS Outposts 或 AWS Local Zones 的指南。

## 概述

Kubernetes 版本包括控制平面和数据平面。为了确保平稳运行，控制平面和数据平面都应运行相同的 Kubernetes 次要版本，例如 1.24。在 AWS 管理和升级控制层面的同时，更新数据平面中的工作节点是您的责任。

- 控制平面 — 控制平面的版本由 Kubernetes API 服务器决定。在 Amazon EKS 集群中，AWS 负责管理此组件。控制平面升级可以通过 AWS API 启动。
- 数据平面 — 数据平面版本与在您的各个节点上运行的 Kubelet 版本相关联。可以让同一个集群中的节点运行不同的版本。您可以通过运行来检查所有节点的版本`kubectl get nodes`。

## 升级之前

如果您计划在 Amazon EKS 中升级 Kubernetes 版本，那么在开始升级之前，您应该制定一些重要的政策、工具和程序。

- 了解弃用政策 — 深入了解 Kubernetes 弃用政策的工作原理。请注意任何可能影响您现有应用程序的即将发生的更改。较新版本的 Kubernetes 通常会逐步淘汰某些 API 和功能，这可能会导致应用程序的运行出现问题。
- 查看 Kubernetes 变更日志 — 彻底查看 Kubernetes 变更日志以及 Amazon EKS Kubernetes 版本，以了解对集群的任何可能影响，例如可能影响工作负载的重大更改。
- 评估集群 Add-Ons 兼容性 — 在新版本发布或将集群更新到新的 Kubernetes 次要版本之后，Amazon EKS 不会自动更新附加组件。查看更新插件，了解任何现有集群插件与您打算升级到的集群版本的兼容性。

- 启用控制平面日志记录-启用控制平面日志记录以捕获升级过程中可能出现的日志、错误或问题。考虑查看这些日志中是否存在任何异常情况。在非生产环境中测试集群升级，或者将自动测试集成到持续集成工作流程中，以评估版本与应用程序、控制器和自定义集成的兼容性。
- 探索 eksctl 进行集群管理 — 考虑使用 eksctl 来管理你的 EKS 集群。它使您能够开箱即用地更新控制平面、管理插件和处理工作节点更新。
- 选择自动模式或托管节点组 — 使用自动模式或 EKS 托管节点组简化和自动执行工作节点升级。这些选项简化了流程并减少了手动干预。
- 利用 kubectl 转换插件 — 利用 kubectl 转换插件来促进不同 API 版本之间的 Kubernetes 清单文件转换。这有助于确保您的配置与新的 Kubernetes 版本保持兼容。

## 让您的集群保持最新状态

及时了解 Kubernetes 更新对于安全高效的 EKS 环境至关重要，这反映了 Amazon EKS 中的分担责任模式。通过将这些策略整合到您的运营工作流程中，您可以将自己定位为维护最新、安全的集群，充分利用最新的功能和改进。战术：

- 支持的版本政策 — 与 Kubernetes 社区保持一致，Amazon EKS 通常提供三个有效的 Kubernetes 版本。Kubernetes 次要版本在发布后的前 14 个月内受亚马逊 EKS 的标准支持。一个版本超过标准支持终止日期后，将在接下来 12 个月自动进入扩展支持。弃用通知在版本到期标准支持日期前至少 60 天发布。有关更多详细信息，请参阅 E KS 版本生命周期文档。
- Auto-Upgrade 政策 — 我们强烈建议您与 EKS 集群中的 Kubernetes 更新保持同步。在已完成 26 个月生命周期（14 个月的标准支持期加上 12 个月的延期支持期）的 Kubernetes 版本上运行的集群，会自动升级到下一个版本。请注意，您可以禁用扩展支持。如果在版本生命周期结束之前未能主动升级，则会触发自动升级，这可能会中断您的工作负载和系统。有关更多信息，请参阅 EKS 版本常见问题解答。
- 创建升级操作手册 — 建立有据可查的升级管理流程。作为主动方法的一部分，开发针对您的升级过程量身定制的运行手册和专业工具。这不仅可以增强您的准备能力，还可以简化复杂的过渡。将每年至少升级一次集群作为标准做法。这种做法使您与持续的技术进步保持一致，从而提高环境的效率和安全性。

## 查看 EKS 发布日历

查看 EKS Kubernetes 发布日历，了解新版本何时推出以及对特定版本的支持何时结束。通常，EKS 每年发布三个 Kubernetes 次要版本，每个次要版本的支持期约为 14 个月。

此外，请查看上游 Kubernetes 版本信息。

# 了解分担责任模式如何适用于集群升级

您负责启动集群控制平面和数据平面的升级。了解如何启动升级。当您启动集群升级时，AWS 会管理升级集群控制平面。你负责升级数据平面，包括 Fargate 吊舱和插件。您必须验证集群上运行的工作负载并计划升级，以确保集群升级后其可用性和操作不会受到影响

## 就地升级集群

EKS 支持就地集群升级策略。这样可以维护群集资源，并保持集群配置的一致性（例如，API 端点、OIDC、ENI、负载均衡器）。这样可以减少对集群用户的干扰，并且无需您重新部署工作负载或迁移外部资源（例如 DNS、存储），即可使用集群中的现有工作负载和资源。

在执行就地集群升级时，请务必注意，一次只能执行一个次要版本升级（例如，从 1.24 到 1.25）。

这意味着，如果您需要更新多个版本，则需要进行一系列的顺序升级。规划顺序升级更为复杂，停机风险也更高。在这种情况下，请参阅the section called “评估 Blue/Green 集群作为就地集群升级的替代方案”。

## 按顺序升级您的控制平面和数据平面

要升级集群，您需要采取以下操作：

1. 查看 Kubernetes 和 EKS 的发行说明。
2. 对集群进行备份。（可选）
3. 识别并修复工作负载中已弃用和已移除的 API 使用情况。
4. 确保托管节点组（如果使用）与控制平面处于相同的 Kubernetes 版本上。EKS 托管节点组和由 EKS Fargate Profiles 创建的节点支持 Kubernetes 1.27 及以下版本控制平面和数据平面之间的 2 个次要版本偏差。从 1.28 及更高版本开始，EKS 托管的节点组和由 EKS Fargate Profiles 创建的节点支持控制平面和数据平面之间的 3 个次要版本偏差。例如，如果你的 EKS 控制平面版本是 1.28，那么你可以放心地使用低至 1.25 的 kubelet 版本。如果你的 EKS 版本是 1.27，那么你可以使用的最旧的 kubelet 版本是 1.25。
5. 使用 AWS 控制台或 cli 升级集群控制平面。
6. 查看附加组件的兼容性。根据需要升级您的 Kubernetes 插件和自定义控制器。
7. 更新 kubectl。
8. 升级集群数据平面。将您的节点升级到与升级后的集群相同的 Kubernetes 次要版本。

> **Tip**
>
> 如果您的集群是使用 EKS 自动模式创建的，则无需升级集群数据平面。升级控制平面后，EKS Auto Mode 将开始逐步更新托管节点，同时考虑所有 pod 中断预算。确保监控这些更新，以验证是否符合您的运营要求。

## 使用 EKS 文档创建升级清单

EKS Kubernetes 版本文档包括每个版本的详细更改列表。为每次升级制定一份清单。

有关特定的 EKS 版本升级指南，请查看文档，了解每个版本的重大更改和注意事项。

- 查看有关标准支持的 Kubernetes 版本的发行说明
- 查看 Kubernetes 版本的发行说明，了解扩展支持

## 使用 Kubernetes API 升级插件和组件

在升级集群之前，您应该了解自己使用的 Kubernetes 组件的版本。清点集群组件，识别直接使用 Kubernetes API 的组件。这包括关键的集群组件，例如监控和日志代理、集群自动扩缩器、容器存储驱动程序（例如 EBS CSI、E F S CSI）、入口控制器以及任何其他直接依赖 Kubernetes API 的工作负载或附加组件。

> **Tip**
>
> 关键集群组件通常安装在`*-system`命名空间中

```
kubectl get ns | grep '-system'
```

确定依赖于 Kubernetes API 的组件后，请查看其文档以了解版本兼容性和升级要求。例如，有关版本兼容性，请参阅 AWS Load Balancer 控制器文档。在继续群集升级之前，可能需要升级某些组件或更改配置。需要检查的一些关键组件包括 CoreDNS、kube-proxy、VPC CNI 和存储驱动程序。

集群通常包含许多使用 Kubernetes API 的工作负载，这些工作负载功能是入口控制器、持续交付系统和监控工具等工作负载功能所必需的。升级 EKS 集群时，还必须升级插件和第三方工具，以确保它们兼容。

请参阅以下常用插件示例及其相关升级文档：

- 亚马逊 VPC CNI：有关每个集群版本的 Amazon VPC CNI 插件的推荐版本，请参阅更新 Kubernetes 自我管理插件的 Amazon VPC CNI 插件。作为 Amazon EKS 安装时 Add-on，一次只能升级一个次要版本。
- kube-proxy：请参阅更新 Kubernetes kub e-proxy 自我管理插件。
- CoreDNS：请参阅更新 CoreDNS 自我管理插件。
- AWS Load Balancer 控制器：AWS Load Balancer 控制器需要与您部署的 EKS 版本兼容。有关更多信息，请参阅安装指南。
- 亚马逊 Elastic Block Store (Amazon EBS) Amazon 容器存储接口 (CSI) 驱动程序：有关安装和升级信息，请参阅作为亚马逊 EKS 插件管理亚马逊 EBS CSI 驱动程序。
- Amazon Elastic File System (Amazon EFS) 容器存储接口 (CSI) 驱动程序：有关安装和升级信息，请参阅亚马逊 EFS CSI 驱动程序。
- Kubernetes 指标服务器：有关更多信息，请参阅上的 metrics-server。 GitHub
- Kubernetes 集群自动扩缩器：要升级 Kubernetes 集群自动扩缩器的版本，请在部署中更改镜像的版本。集群自动扩缩程序与 Kubernetes 调度器紧密结合。升级集群时，您始终需要对其进行升级。查看GitHub 发行版以查找与您的 Kubernetes 次要版本相对应的最新版本的地址。
- Karpenter：有关安装和升级信息，请参阅 Kar penter 文档。

> **Tip**
>
> 您无需手动升级 Amazon EKS 自动模式的任何功能，包括计算自动扩展、块存储和负载平衡功能。

# 升级前验证 EKS 的基本要求

AWS 需要您的账户中的某些资源才能完成升级过程。如果这些资源不存在，则无法升级集群。控制平面升级需要以下资源：

1. 可用 IP 地址：Amazon EKS 需要您在创建集群时指定的子网中最多五个可用 IP 地址才能更新集群。如果不是，请在执行版本更新之前更新您的集群配置以包含新的集群子网。
2. EKS IAM 角色：控制平面 IAM 角色仍存在于具有必要权限的账户中。
3. 如果您的集群启用了秘密加密，请确保集群 IAM 角色有权使用 AWS Key Management Service (AWS KMS) 密钥。

## 验证可用的 IP 地址

为了升级集群，Amazon EKS 需要您在创建集群时指定的子网中的最多 5 个可用的 IP 地址。

要验证您的子网是否有足够的 IP 地址来升级集群，可以运行以下命令：

```
CLUSTER=<cluster name>
aws ec2 describe-subnets --subnet-ids \
  $(aws eks describe-cluster --name ${CLUSTER} \
  --query 'cluster.resourcesVpcConfig.subnetIds' \
  --output text) \
  --query 'Subnets[*].[SubnetId,AvailabilityZone,AvailableIpAddressCount]' \
  --output table

-------------------------------------------------------
|                  DescribeSubnets                    |
+---------------------------+--------------+-------+
|  subnet-067fa8ee8476abbd6 |  us-east-1a  |  8184 |
|  subnet-0056f7403b17d2b43 |  us-east-1b  |  8153 |
|  subnet-09586f8fb3addbc8c |  us-east-1a  |  8120 |
|  subnet-047f3d276a22c6bce |  us-east-1b  |  8184 |
+---------------------------+--------------+-------+
```

VPC CNI 指标助手可用于为 VPC 指标创建 CloudWatch 控制面板。如果您在创建集群时最初指定的子网中的 IP 地址已用完，Amazon EKS 建议您在开始 Kubernetes 版本升级之前使用 UpdateClusterConfiguration “” API 更新集群子网。请确认将为您提供新子网：

- 属于在创建集群时选择的同一组可用区。
- 属于创建集群时提供的相同 VPC

如果现有 VPC CIDR 区块中的 IP 地址用完，请考虑关联其他 CIDR 块。AWS 支持将更多 CIDR 块与您的现有集群 VPC 关联，从而有效地扩展您的 IP 地址池。这种扩展可以通过引入额外的私有 IP 范围（RFC 1918）或在必要时引入公有 IP 范围（非 RFC 1918）来实现。在 Amazon EKS 可以使用新的 CIDR 之前，您必须添加新的 VPC CIDR 块并允许 VPC 刷新完成。之后，您可以根据新设置的 CIDR 块将子网更新到 VPC。

## 验证 EKS 的 IAM 角色

要验证 IAM 角色是否可用并且您的账户中是否具有正确的代入角色策略，您可以运行以下命令：

```
CLUSTER=<cluster name>
```

```
ROLE_ARN=$(aws eks describe-cluster --name ${CLUSTER} \
  --query 'cluster.roleArn' --output text)
aws iam get-role --role-name ${ROLE_ARN##*/} \
  --query 'Role.AssumeRolePolicyDocument'

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

# 迁移到 EKS Add-ons

Amazon EKS 会自动安装附加组件，例如适用于 Kubernetes 的 Amazon VPC CNI 插件，kube-proxy以及每个集群的 CoreDNS。 Add-ons 可以自行管理，也可以作为 Amazon EKS Add-ons 安装。Amazon EKS Add-ons 是使用 EKS API 管理附加组件的另一种方式。

您可以使用 Amazon EKS Add-ons 通过单个命令更新版本。例如：

```
aws eks update-addon –cluster-name my-cluster –addon-name vpc-cni –addon-version
 version-number \
--service-account-role-arn arn:aws:iam::111122223333:role/role-name –configuration-
values '{}' –resolve-conflicts PRESERVE
```

检查你是否有 Add-ons 带有以下内容的 EKS：

```
aws eks list-addons --cluster-name <cluster name>
```

> **Warning**
>
> Add-ons 在控制平面升级期间，EKS 不会自动升级。您必须启动 EKS 插件更新，然后选择所需的版本。

- 您有责任从所有可用版本中选择兼容版本。查看有关附加组件版本兼容性的指南。
- Amazon EKS 一次 Add-ons 只能升级一个次要版本。

详细了解哪些组件可用作 EKS Add-ons 以及如何入门。

了解如何向 EKS 提供自定义配置 Add-on。

# 在升级控制平面之前，识别并修复已删除的 API 使用情况

在升级 EKS 控制平面之前，您应该确定已移除的 API 的 API 使用情况。为此，我们建议使用可以检查正在运行的集群或静态渲染的 Kubernetes 清单文件的工具。

对静态清单文件进行检查通常更准确。如果针对实时集群运行，这些工具可能会返回误报。

已弃用的 Kubernetes API 并不意味着该 API 已被删除。你应该查看 Kubernetes 弃用政策，了解移除 API 会如何影响你的工作负载。

## 集群见解

Clu@@ ster Insights 是一项功能，可提供有关可能影响 EKS 集群升级到较新版本的 Kubernetes 能力的问题的调查结果。这些发现由 Amazon EKS 策划和管理，并就如何补救提供了建议。通过利用集群见解，您可以最大限度地减少升级到较新 Kubernetes 版本所花费的工作量。

要查看 EKS 集群的见解，你可以运行以下命令：

```
aws eks list-insights --region <region-code> --cluster-name <my-cluster>

{
    "insights": [
        {
            "category": "UPGRADE_READINESS",
            "name": "Deprecated APIs removed in Kubernetes v1.29",
            "insightStatus": {
                "status": "PASSING",
                "reason": "No deprecated API usage detected within the last 30 days."
            },
            "kubernetesVersion": "1.29",
            "lastTransitionTime": 1698774710.0,
            "lastRefreshTime": 1700157422.0,
            "id": "123e4567-e89b-42d3-a456-579642341238",
```

```
            "description": "Checks for usage of deprecated APIs that are scheduled for
 removal in Kubernetes v1.29. Upgrading your cluster before migrating to the updated
 APIs supported by v1.29 could cause application impact."
        }
    ]
}
```

要获得有关收到的见解的更具描述性的输出，可以运行以下命令：

```
aws eks describe-insight --region <region-code> --id <insight-id> --cluster-name <my-
cluster>
```

您还可以选择在 Amazon EKS 控制台中查看见解。从集群列表中选择您的集群后，见解结果将位于Upgrade Insights选项卡下方。

如果您发现了有关集群的见解"status": ERROR，则必须在执行集群升级之前解决该问题。运行将共享以下补救建议的aws eks describe-insight命令：

受影响的资源：

```
"resources": [
      {
        "insightStatus": {
          "status": "ERROR"
        },
        "kubernetesResourceUri": "/apis/policy/v1beta1/podsecuritypolicies/null"
      }
]
```

已弃用的 API：

```
"deprecationDetails": [
      {
        "usage": "/apis/flowcontrol.apiserver.k8s.io/v1beta2/flowschemas",
        "replacedWith": "/apis/flowcontrol.apiserver.k8s.io/v1beta3/flowschemas",
        "stopServingVersion": "1.29",
        "clientStats": [],
        "startServingReplacementVersion": "1.26"
      }
]
```

建议采取的行动：

```
"recommendation": "Update manifests and API clients to use newer Kubernetes APIs if
 applicable before upgrading to Kubernetes v1.26."
```

通过 EKS 控制台或 CLI 利用集群见解有助于加快成功升级 EKS 集群版本的过程。通过以下资源了解更多信息：* EKS 官方 Docs * C l uster Insights 发布博客。

## Kube-no-trouble

Kube-no-trouble是一个带有命令的开源命令行实用程序kubent。当你在没有任何参数kubent的情况下运行时，它将使用你当前的 KubeConfig上下文扫描集群并打印一份报告，其中包含哪些将被弃用和移除的 API。

```
kubent

4:17PM INF >>> Kube No Trouble `kubent` <<<
4:17PM INF version 0.7.0 (git sha d1bb4e5fd6550b533b2013671aa8419d923ee042)
4:17PM INF Initializing collectors and retrieving data
4:17PM INF Target K8s version is 1.24.8-eks-ffeb93d
4:l INF Retrieved 93 resources from collector name=Cluster
4:17PM INF Retrieved 16 resources from collector name="Helm v3"
4:17PM INF Loaded ruleset name=custom.rego.tmpl
4:17PM INF Loaded ruleset name=deprecated-1-16.rego
4:17PM INF Loaded ruleset name=deprecated-1-22.rego
4:17PM INF Loaded ruleset name=deprecated-1-25.rego
4:17PM INF Loaded ruleset name=deprecated-1-26.rego
4:17PM INF Loaded ruleset name=deprecated-future.rego
_____________________________________________________________________________________
>>> Deprecated APIs removed in 1.25 <<<
-------------------------------------------------------------------------------------
KIND                NAMESPACE     NAME             API_VERSION      REPLACE_WITH
 (SINCE)
PodSecurityPolicy   <undefined>   eks.privileged   policy/v1beta1   <removed> (1.21.0)
```

它还可用于扫描静态清单文件和 helm 软件包。建议将其kubent作为持续集成 (CI) 流程的一部分运行，以便在清单部署之前识别问题。扫描清单也比扫描实时集群更准确。

Kube-no-trouble 提供了一个示例服务帐户和角色，该帐户和角色具有扫描集群的相应权限。

## 冥王星

另一种选择是 p lut o，它之所以类似，kubent是因为它支持扫描实时集群、清单文件、头盔图表，并且有一个可以包含在 CI 流程中的 GitHub 操作。

```
pluto detect-all-in-cluster

NAME             KIND                VERSION          REPLACEMENT   REMOVED
 DEPRECATED   REPL AVAIL
eks.privileged   PodSecurityPolicy   policy/v1beta1                 false     true
    true
```

## 资源

要在升级之前验证您的集群是否未使用已弃用的 API，您应该监控：

- apiserver_requested_deprecated_apis自 Kubernetes v1.19 以来的指标：

```
kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis

apiserver_requested_deprecated_apis{group="policy",removed_release="1.25",resource="podsecurity
 1
```

- 审核日志中的事件k8s.io/deprecated设置为true:

```
CLUSTER="<cluster_name>"
QUERY_ID=$(aws logs start-query \
 --log-group-name /aws/eks/${CLUSTER}/cluster \
 --start-time $(date -u --date="-30 minutes" "+%s") # or date -v-30M "+%s" on MacOS \
 --end-time $(date "+%s") \
 --query-string 'fields @message | filter `annotations.k8s.io/deprecated`="true"' \
 --query queryId --output text)

echo "Query started (query id: $QUERY_ID), please hold ..." && sleep 5 # give it some
 time to query

aws logs get-query-results --query-id $QUERY_ID
```

如果正在使用已弃用的 API，则会输出行：

```
{
    "results": [
        [
            {
                "field": "@message",
                "value": "{\"kind\":\"Event\",\"apiVersion\":\"audit.k8s.io/v1\",
\"level\":\"Request\",\"auditID\":\"8f7883c6-b3d5-42d7-967a-1121c6f22f01\",\"stage
\":\"ResponseComplete\",\"requestURI\":\"/apis/policy/v1beta1/podsecuritypolicies?
allowWatchBookmarks=true\\u0026resourceVersion=4131\\u0026timeout=9m19s\
\u0026timeoutSeconds=559\\u0026watch=true\",\"verb\":\"watch\",\"user\":{\"username
\":\"system:apiserver\",\"uid\":\"8aabfade-da52-47da-83b4-46b16cab30fa\",
\"groups\":[\"system:masters\"]},\"sourceIPs\":[\"::1\"],\"userAgent\":\"kube-
apiserver/v1.24.16 (linux/amd64) kubernetes/af930c1\",\"objectRef\":{\"resource
\":\"podsecuritypolicies\",\"apiGroup\":\"policy\",\"apiVersion\":\"v1beta1\"},
\"responseStatus\":{\"metadata\":{},\"code\":200},\"requestReceivedTimestamp\":
\"2023-10-04T12:36:11.849075Z\",\"stageTimestamp\":\"2023-10-04T12:45:30.850483Z\",
\"annotations\":{\"authorization.k8s.io/decision\":\"allow\",\"authorization.k8s.io/
reason\":\"\",\"k8s.io/deprecated\":\"true\",\"k8s.io/removed-release\":\"1.25\"}}"
            },
[...]
```

# 更新 Kubernetes 工作负载。使用 kubectl-convert 更新清单

确定需要更新哪些工作负载和清单后，可能需要将清单文件中的资源类型更改为（例如，更改 PodSecurityPolicies 为 PodSecurityStandards）。这将需要更新资源规格并根据要替换的资源进行额外的研究。

如果资源类型保持不变，但需要更新 API 版本，则可以使用`kubectl-convert`命令自动转换清单文件。例如，将较旧的部署转换为`apps/v1`。有关更多信息，请参阅在 Kubernetes 网站上安装 kubectl 转换插件。

```
kubectl-convert -f <file> --output-version <group>/<version>
```

# 配置 PodDisruptionBudgets 和拓扑结构SpreadConstraints 以确保升级数据平面时工作负载的可用性

确保您的工作负载具有适当的PodDisruptionBudgets拓扑结构 SpreadConstraints，以确保在数据平面升级期间工作负载的可用性。并非每个工作负载都需要相同的可用性级别，因此您需要验证工作负载的规模和要求。

确保工作负载分布在多个可用区中，并且在具有拓扑分布的多台主机上，这样可以更有信心工作负载会自动迁移到新的数据平面，而不会发生任何事故。

以下是一个工作负载示例，该工作负载将始终有 80% 的副本可用，并将副本分布在各个区域和主机上

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: myapp
spec:
  minAvailable: "80%"
  selector:
    matchLabels:
      app: myapp
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 10
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: public.ecr.aws/eks-distro/kubernetes/pause:3.2
        name: myapp
        resources:
          requests:
            cpu: "1"
            memory: 256M
      topologySpreadConstraints:
      - labelSelector:
          matchLabels:
            app: host-zone-spread
        maxSkew: 2
        topologyKey: kubernetes.io/hostname
        whenUnsatisfiable: DoNotSchedule
      - labelSelector:
```

```
      matchLabels:
        app: host-zone-spread
    maxSkew: 2
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
```

AWS Resilience Hub 已将亚马逊 Elastic Kubernetes Service（Amazon EKS）添加为支持资源。Resilience Hub 提供了一个定义、验证和跟踪应用程序弹性的单一位置，这样您就可以避免因软件、基础设施或运营中断而导致的不必要停机。

## 使用托管节点组或 Karpenter 来简化数据平面升级

托管节点组和 Karpenter 都简化了节点升级，但它们采用了不同的方法。

托管节点组可自动执行节点的配置和生命周期管理。这意味着您可以通过单个操作创建、自动更新或终止节点。

在默认配置中，Karpenter 使用最新兼容的 EKS 优化 AMI 自动创建新节点。当 EKS 发布更新的 EKS 优化 AMI 或集群升级时，Karpenter 将自动开始使用这些映像。Karpenter 还实现了节点过期来更新节点。

可以将 Karpenter 配置为使用自定义 AMI。如果您在 Karpenter 中使用自定义 AMI，则您需要对 kubelet 的版本负责。

## 确认版本与现有节点和控制平面的兼容性

在 Amazon EKS 中继续升级 Kubernetes 之前，确保托管节点组、自我管理节点和控制平面之间的兼容性至关重要。兼容性由你使用的 Kubernetes 版本决定，它因不同的场景而异。战术：

- Kubernetes v1.28+ — * * 从 Kubernetes 版本 1.28 及更高版本开始，核心组件的版本政策更加宽松。具体而言，Kubernetes API 服务器和 kubelet 之间支持的偏差已扩展了一个次要版本，从 n-2 扩展到 n-3。例如，如果你的 EKS 控制平面版本是 1.28，那么你可以放心地使用低至 1.25 的 kubelet 版本。AW S Fargate 、托管节点组和自管理节点均支持此版本偏差。出于安全考虑，我们强烈建议您的 Amazon 系统映像 (AMI) 版本保持最新。由于潜在的常见漏洞和暴露 (CVE)，较旧的 kubelet 版本可能会带来安全风险，这可能超过使用旧版 kubelet 的好处。
- Kubernetes < v1.28 — 如果你使用的是早于 1.28 的版本，那么 API 服务器和 kubelet 之间支持的偏差为 n-2。例如，如果你的 EKS 版本是 1.27，那么你可以使用的最旧的 kubelet 版本是 1.25。此版本偏差适用于 AWS Fargate 、托管节点组和自管理节点。

# 为 Karpenter 托管的节点启用节点到期时间

Karpenter 实现节点升级的一种方法是使用节点过期的概念。这减少了节点升级所需的规划。当您在配置器SecondsUntilExpired 中为 ttl 设置值时，这会激活节点过期时间。节点在几秒钟内达到定义的年龄后，它们就会被安全地耗尽并删除。即使它们正在使用也是如此，允许您将节点替换为新配置的升级实例。当节点被替换时，Karpenter 会使用最新 EKS-optimized 的 AMI。有关更多信息，请参阅 Karpenter 网站上的 “取消配置”。

Karpenter 不会自动为该值添加抖动。为防止工作负载过度中断，请定义 Pod 中断预算，如 Kubernetes 文档中所示。

如果您在置备器SecondsUntilExpired 上配置 ttl，则这适用于与置备程序关联的现有节点。

# 对 Karpenter 托管的节点使用漂移功能

Karpenter 的 Drift 功能可以自动升级 Karpenter-provisioned 节点，使其与 EKS 控制平面保持同步。目前需要使用功能门启用 Karpenter Drift。Karpenter 的默认配置使用与 EKS 集群控制平面相同的主版本和次要版本的最新 EKS-Optimized AMI。

EKS 集群升级完成后，Karpenter 的 Drift 功能将检测到 Karpenter-provisioned 节点正在使用先前集群版本 EKS-Optimized 的 AMI，并自动封锁、耗尽和替换这些节点。要支持 Pod 迁移到新节点，请通过设置适当的 pod 资源配额和使用 Pod 中断预算 (PDB) 来遵循 Kubernetes 最佳实践。Karpenter 的取消配置将根据 pod 资源请求预先启动替换节点，并且在取消配置节点时将尊重 PDB。

# 使用 eksctl 自动升级自管理节点组

自我管理节点组是部署在您的账户中并连接到 EKS 服务之外的集群的 EC2 实例。它们通常由某种形式的自动化工具进行部署和管理。要升级自行管理的节点组，您应该参考您的工具文档。

例如，eksctl 支持删除和耗尽自管理节点。

一些常用工具包括：

- eksctl
- KOP
- EKS 蓝图

# 升级前备份集群

新版本的 Kubernetes 对您的 Amazon EKS 集群进行了重大更改。升级集群后，您无法对其进行降级。

Velero 是一个社区支持的开源工具，可用于备份现有集群并将备份应用到新集群。

请注意，您只能为 EKS 目前支持的 Kubernetes 版本创建新的集群。如果您的集群当前运行的版本仍然受支持，并且升级失败，则可以使用原始版本创建新集群并恢复数据平面。请注意，包括 IAM 在内的 AWS 资源不包含在 Velero 的备份中。这些资源需要重新创建。

# 升级控制平面后重启 Fargate 部署

要升级 Fargate 数据平面节点，您需要重新部署工作负载。您可以通过列出带有`-o wide`选项的所有 pod 来识别哪些工作负载在 fargate 节点上运行。任何以开头的节点名称都`fargate-`需要在集群中重新部署。

# 评估 Blue/Green 集群作为就地集群升级的替代方案

有些客户更喜欢采取 blue/green 升级策略。这可能有好处，但也包括应考虑的缺点。

优势包括：

- 可以同时更改多个 EKS 版本（例如 1.23 到 1.25）
- 能够切换回旧集群
- 创建一个可以用较新的系统（例如 terraform）管理的新集群
- 工作负载可以单独迁移

一些缺点包括：

- API 端点和 OIDC 变更需要更新使用者（例如 kubectl 和） CI/CD
- 迁移期间需要 2 个群集并行运行，这可能很昂贵且会限制区域容量
- 如果工作负载相互依赖才能一起迁移，则需要加强协调
- 负载均衡器和外部 DNS 无法轻松跨越多个集群

虽然这种策略是可能的，但它比就地升级更昂贵，而且需要更多的时间进行协调和工作负载迁移。在某些情况下可能需要这样做，并且应谨慎规划。

有了高度的自动化和像这样的声明式系统 GitOps，这可能更容易做到。对于有状态的工作负载，您需要采取额外的预防措施，以便备份数据并将其迁移到新集群。

查看以下博客文章以获取更多信息：

- Kubernetes 集群升级：蓝绿部署策略
- Blue/Green或针对无状态 ArgoCD 工作负载的 Canary Amazon EKS 集群

# 跟踪 Kubernetes 项目中计划中的重大变更——提前思考

不要只看下一个版本。在 Kubernetes 的新版本发布时对其进行审查，并确定重大变化。例如，一些应用程序直接使用 docker API，而 Kubernetes 中删除了对 Docker（也称为 Dockershim）的容器运行时接口 (CRI) 的支持。1.24这种变化需要更多的时间来做好准备。

查看您要升级到的版本的所有已记录的更改，并记下所有必需的升级步骤。此外，请注意特定于 Amazon EKS 托管集群的任何要求或程序。

- Kubernetes 更新日志

# 移除功能的具体指南

## 在 1.25 中移除 Dockershim ——在 Docker 插槽中使用探测器 (DDS)

适用于 1.25 的 EKS 优化 AMI 不再包括对 Dockershim 的支持。如果你依赖于 Dockershim，例如你正在安装 Docker 套接字，那么在将工作节点升级到 1.25 之前，你需要删除这些依赖关系。

在升级到 1.25 之前，请查找依赖于 Docker 套接字的实例。我们建议使用适用于 Docker Socket 的探测器 (DDS)，这是一个 kub ectl 插件。 。

## PodSecurityPolicy 在 1.25 中移除-迁移到 Pod 安全标准或策略即代码解决方案

PodSecurityPolicy在 Kubernetes 1.21 中已被弃用，在 Kubernetes 1. 25 中已被删除。如果您在集群 PodSecurityPolicy 中使用，则必须先迁移到内置的 Kubernetes Pod 安全标准 (PSS) 或策略即代码解决方案，然后再将集群升级到版本 1.25，以避免工作负载中断。

AWS 在 EKS 文档中发布了详细的常见问题解答。

查看 Pod 安全标准 (PSS) 和 Pod 安全准入 (PSA) 最佳实践。

查看 Kubernetes 网站上的 “PodSecurityPolicy弃用” 博客文章。

### 1.23 中弃用 In-Tree 存储驱动程序-迁移到容器存储接口 (CSI) 驱动程序

容器存储接口 (CSI) 旨在帮助 Kubernetes 取代其现有的树内存储驱动程序机制。默认情况下，Amazon EBS Container Storage Interface (CSI) 迁移功能在 Amazon EKS `1.23` 和更高版本的集群上处于启用状态。如果您在某个版本`1.22`或更早版本的集群上运行 Pod，则必须先安装 Amazon EBS CSI 驱动程序，然后再将集群更新到版本，`1.23`以避免服务中断。

查看 Amazon EBS CSI 迁移常见问题解答。

## 其他资源

### ClowdHaus EKS 升级指南

ClowdHaus EKS 升级指南是一个 CLI，用于帮助升级 Amazon EKS 集群。它可以在升级之前分析集群中是否存在任何需要修复的潜在问题。

### GoNoGo

GoNoGo是一款 alpha 阶段工具，用于确定集群插件的升级可信度。

# 成本优化的最佳实践

成本优化就是以最低的价格实现您的业务成果。按照本指南中的文档进行操作，您将优化您的 Amazon EKS 工作负载。

## 一般指导原则

在云中，有许多通用准则可以帮助您实现微服务的成本优化：

- 确保在 Amazon EKS 上运行的工作负载独立于用于运行容器的特定基础设施类型，这将为在最便宜的基础设施类型上运行它们提供更大的灵活性。在将 Amazon EKS 与配合使用时 EC2，由于工作负载的性质，当我们的工作负载需要特定类型的 EC2 实例类型（例如需要 GPU 或其他实例类型）时，可能会出现例外情况。
- 选择经过优化配置的容器实例 — 使用适用于 Amazon EKS 的 Ama CloudWatch zon Container Insights 或 Kubernetes 生态系统中提供的第三方工具，分析您的生产或预生产环境，并监控 CPU 和内存等关键指标。这将确保我们可以分配适当数量的资源，避免浪费资源。
- 利用 AWS 中提供的不同购买选项来运行 EKS EC2，例如按需、Spot 和 Savings Plan。

## EKS 成本优化最佳实践

云端成本优化有三个一般最佳实践领域：

- 经济实惠的资源（Auto Scaling、向下扩展、政策和购买选项）
- 支出意识（使用 AWS 和第三方工具）
- 随着时间的推移进行优化（正确尺寸）

与任何指导一样，需要权衡取舍。确保与您的组织合作，了解此工作负载的优先级以及哪些最佳实践最重要。

### 如何使用本指南

本指南适用于负责实施和管理 EKS 集群及其支持的工作负载的开发运营团队。该指南分为不同的最佳实践领域，便于使用。每个主题都列出了 EKS 集群成本优化的建议、要使用的工具和最佳实践。主题无需按特定顺序阅读。

## AWS 服务和 Kubernetes 的主要功能

以下 AWS 服务和功能支持成本优化：

- EC2 实例类型，Savings Plan（和预留实例）和竞价型实例，价格不同。
- Auto Scaling 以及 Kubernetes 原生 Auto Scaling 策略。对于可预测的工作负载，可以考虑使用 Savings Plan（以前的预留实例）。使用托管数据存储（如 EBS 和 EFS），以获得应用程序数据的弹性和持久性。
- 账单和成本管理控制台控制面板以及 AWS Cost Explorer 可概述您的 AWS 使用情况。使用 AWS Organizations 获取详细的账单详情。还分享了几种第三方工具的详细信息。
- Amazon CloudWatch 容器指标提供了有关 EKS 集群资源使用情况的指标。除了 Kubernetes 仪表板之外，Kubernetes 生态系统中还有一些工具可以用来减少浪费。

本指南包括一组建议，您可以使用这些建议来改进 Amazon EKS 集群的成本优化。

# 反馈

本指南发布的目的是收集来自更广泛 GitHub 的 EKS/Kubernetes 社区的直接反馈和建议。如果您认为我们应该在指南中包含最佳实践，请在 GitHub 存储库中提交问题或提交 PR。我们的意图是在服务中添加新功能或出现新的最佳实践时，定期更新指南。

# 成本优化框架

AWS 云经济学是一门通过采用 Amazon EKS 等现代计算技术来帮助客户提高效率和降低成本的学科。该学科建议遵循一种名为 “云财务管理（CFM）框架” 的方法，该框架由四个支柱组成：




## See 支柱：衡量和问责制

See 支柱是一组基础活动和技术，用于定义如何衡量、监控和创建云支出的问责制。它通常被称为 “可观测性”、“仪器” 或 “遥测”。“可观测性” 基础架构的功能和局限性决定了哪些可以优化。清晰地了解成本是成本优化的关键第一步，因为您需要知道自己的起点。这种可见性还将指导您为进一步优化环境而需要进行的活动类型。

以下是我们在 See 支柱方面的最佳实践的简要概述：

- 为您的工作负载定义和维护标签策略。
  - 使用实例标记，通过标记 EKS 集群，您可以查看单个集群的成本，并在成本和使用情况报告中进行分配。
- 使用 K ubecost 等技术建立对 EK S 使用情况的报告和监控。
  - 启用 Cloud Intelligence 仪表板，通过正确标记资源并使用可视化效果，您可以衡量和估算成本。
- 将云成本分配给应用程序、业务线 (LoBs) 和收入流。
- 定义、衡量并 efficiency/value KPIs 与业务利益相关者进行沟通。例如，创建一个 “单位指标” KPI 来衡量每笔交易的成本，例如，拼车服务可能有 “每次行程成本” 的 KPI。

有关与该支柱相关的推荐技术和活动的更多详细信息，请参阅本指南的 “成本优化-可观察性” 部分。

## 节省支柱：成本优化

该支柱基于 “See” 支柱中开发的技术和能力。以下活动通常属于这一支柱：

- 识别并消除环境中的浪费。
- 建筑师和设计旨在提高成本效益。
- 选择最佳购买选项，例如按需实例与竞价型实例。
- 随着服务的发展而适应：随着 AWS 服务的发展，高效使用这些服务的方式可能会发生变化。愿意适应这些变化。

由于这些活动是可操作的，因此它们在很大程度上取决于您的环境特征。问问自己，成本的主要驱动因素是什么？ 您的不同环境提供了哪些商业价值？ 哪些购买选项和基础架构选择（例如实例系列类型）最适合每种环境？

以下是按优先顺序排列的 EKS 集群最常见的成本驱动因素列表：

1. 计算成本：需要仔细考虑组合多种类型的实例系列、购买选项以及在可扩展性与可用性之间取得平衡。有关更多信息，请参阅本指南成本优化-计算部分中的建议。
2. 网络成本：对 EKS 集群使用 3 AZs 可能会增加可用区间的流量成本。有关如何平衡高可用性要求和降低网络流量成本的建议，请参阅本指南的 “成本优化-联网” 部分。
3. 存储成本：根据 EKS 集群中工作负载的 stateful/stateless 性质以及不同存储类型的使用方式，可以将存储视为工作负载的一部分。有关 EKS 存储成本的注意事项，请参阅本指南的 “成本优化-存储” 部分。

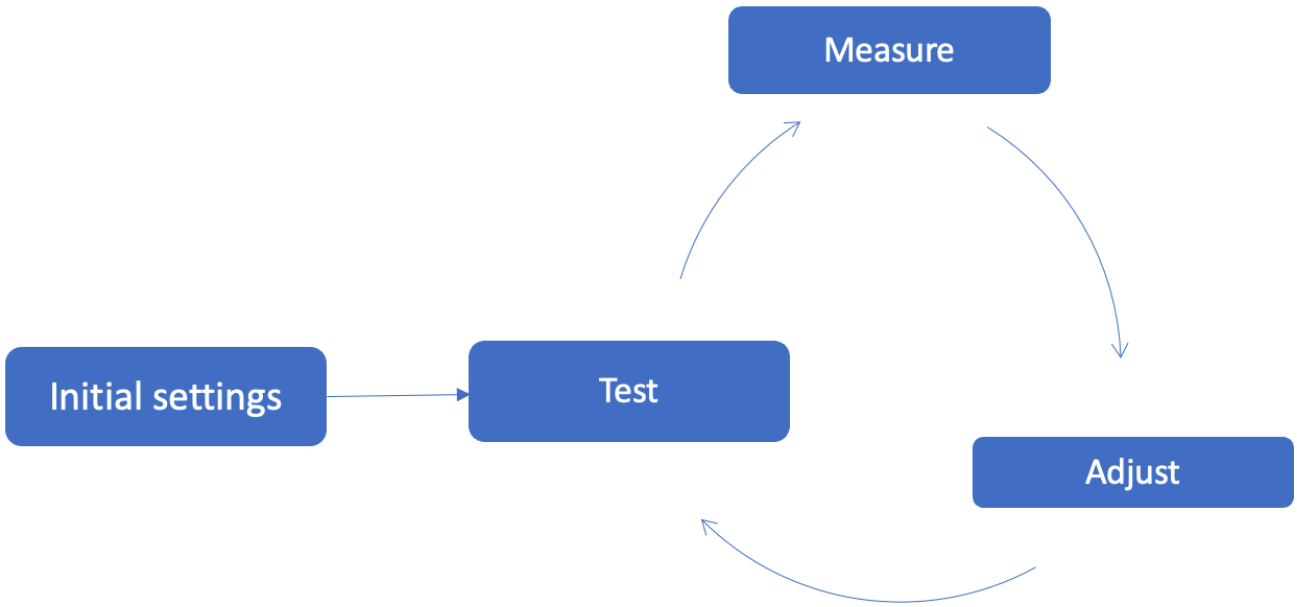
## 计划支柱：规划和预测

一旦 See 支柱中的建议得到实施，集群就会持续优化。随着在高效运营集群方面积累的经验，规划和预测活动可以侧重于：

- 动态预算和预测云成本。
- 量化 EKS 容器服务提供的商业价值。
- 将 EKS 集群成本管理与 IT 财务管理规划相结合。

## 跑步支柱

成本优化是一个持续的过程，涉及逐步改进的飞轮：




确保高管对这类活动的赞助对于将EKS集群优化整合到组织的 “FinOps” 工作中至关重要。它允许利益相关者通过对EKS集群成本的共同理解、实施EKS集群成本护栏以及确保工具、自动化和活动随着组织的需求而发展，来协调利益相关者。

## 参考

- AWS 云经济、云财务管理

# 支出意识

支出意识是了解谁在哪里以及是什么原因导致了您的 EKS 集群中的支出。准确了解这些数据将有助于提高人们对支出的认识，并突出需要补救的领域。

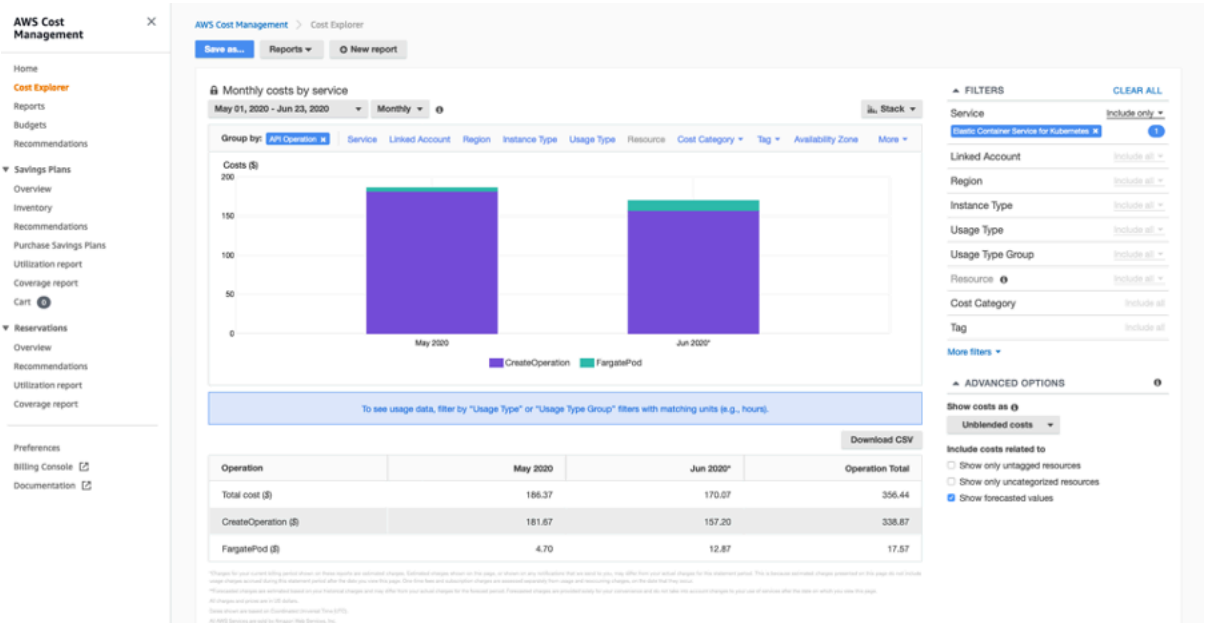
## 建议

### 使用 Cost Explorer

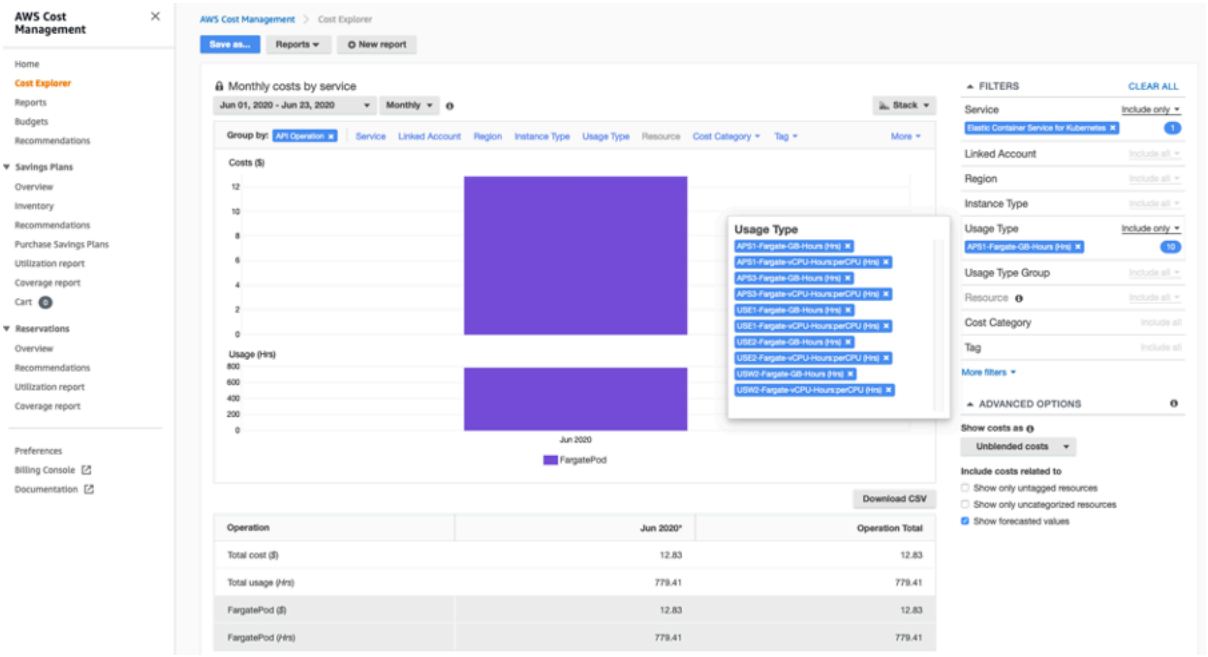
AWS Cost Explorer 有一个 easy-to-use界面，可让您直观地了解、了解和管理您的 AWS 成本和使用情况。您可以使用 Cost Explorer 中提供的筛选器分析不同级别的成本和使用数据。

EKS 控制飞机和 EKS Fargate 成本

使用筛选器，我们可以查询控制平面和 Fargate Pod 的 EKS 成本所产生的成本，如下图所示：




使用筛选器，我们可以查询 EKS 中各个区域的 Fargate Pod 产生的总成本，其中包括每个 CPU 的 vCPU 小时数和 GB 小时数，如下图所示：




为资源加标签

Amazon EK S 支持向您的亚马逊 EKS 集群添加 AWS 标签。这样可以轻松控制对用于管理集群的 EKS API 的访问权限。添加到 EKS 集群的标签特定于 AWS EKS 集群资源，它们不会传播到集群使

用的其他 AWS 资源，例如 EC2 实例或负载均衡器。如今，所有新的和现有的 EKS 集群都支持通过 AWS API、控制台和 SDKs进行集群标记。

AWS Fargate 是一项为容器提供按需、大小合适的计算容量的技术。您必须先定义至少一个 Fargate 配置文件，以指定 Pod 在启动时哪些 Pod 应该使用 Fargate，然后才能安排在集群中的 Fargate 上运行的 Pod。

向 EKS 集群添加和列出标签：

```
$ aws eks tag-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/ekscluster1 --
tags team=devops,env=staging,bu=cio,costcenter=1234
$ aws eks list-tags-for-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/
ekscluster1
{
    "tags": {
        "bu": "cio",
        "env": "staging",
        "costcenter": "1234",
        "team": "devops"
    }
}
```

在 AWS Cost Explorer 中激活成本分配标签后，AWS 会使用成本分配标签在成本分配报告中整理资源成本，以便您更轻松地对自己的 AWS 成本进行分类和跟踪。

标签对 Amazon EKS 没有任何语义意义，应严格按字符串进行解析。例如，您可以为 Amazon EKS 集群定义一组标签，以帮助您跟踪每个集群的拥有者和堆栈级别。

## 使用 AWS Trusted Advisor

AWS Trusted Advisor 提供了五个类别的一系列丰富的最佳实践检查和建议：成本优化、安全、容错、性能和服务限制。

为了实现成本优化，Trusted Advisor 可帮助消除未使用和闲置的资源，并建议对预留容量做出承诺。有助于 Amazon EKS 的关键行动项目将围绕低利用率 EC2 实例、未关联的弹性 IP 地址、空闲负载均衡器、未充分利用的 EBS 卷等。完整的检查清单可在-practice https://aws.amazon.com/premiumsupport/ technology/trusted-advisor/best-checklist/ 中找到。

Trusted Advisor 还为 EC2 实例和 Fargate 提供储蓄计划和预留实例建议，允许您承诺稳定的使用量以换取折扣费率。

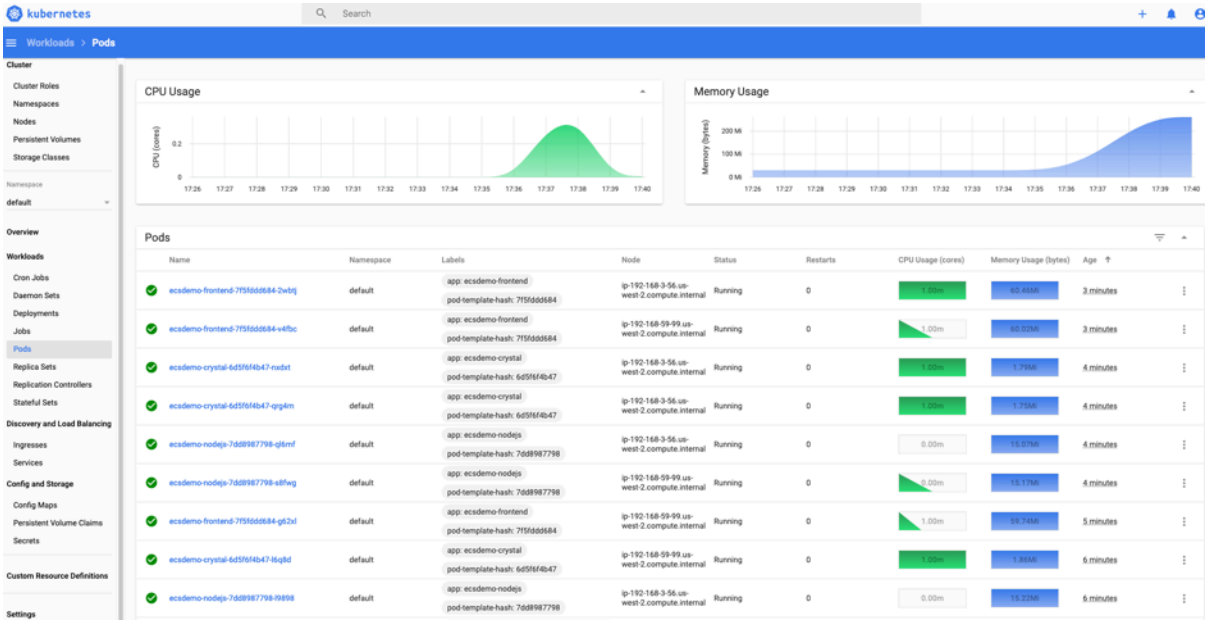
> **Note**
>
> 来自 Trusted Advisor 的建议是一般性建议，并不特定于 EKS。

## 使用 Kubernetes 控制面板

### Kubernetes 控制面板

Kubernetes 控制面板是一个适用于 Kubernetes 集群的通用基于 Web 的用户界面，它提供有关 Kubernetes 集群的信息，包括集群、节点和容器级别的资源使用情况。[亚马逊 EKS 文档中描述了在亚马逊 EKS 集群上部署 Kubernetes 控制面板的情况。]

控制面板提供每个节点和 Pod 的资源使用情况明细，以及有关 Pod、服务、部署和其他 Kubernetes 对象的详细元数据。这些整合的信息提供了对您的 Kubernetes 环境的可见性。




### kubectl top 然后描述命令

使用 kubectl top 和 kubectl describe 命令查看资源使用指标。kubectl top 将显示集群中容器或节点或特定容器或节点的当前 CPU 和内存使用情况。kubectl describe 命令将提供有关特定节点或 Pod 的更多详细信息。

```
$ kubectl top pods
$ kubectl top nodes
$ kubectl top pod pod-name --namespace mynamespace --containers
```

使用 top 命令，输出将显示节点正在使用的 CPU（以内核为单位）和内存（以 MiB 为单位）的总量，以及这些数字所代表的节点可分配容量的百分比。然后，通过添加 --c ontainers标志，你可以深入到容器中的下一个级别，即容器级别。

```
$ kubectl describe node <node>
$ kubectl describe pod <pod>
```

kubectl des cribe 返回每个资源请求或限制所代表的总可用容量的百分比。

kubectl 在 kubernetes pod、节点和容器中描述、跟踪关键资源（例如 CPU、内存和存储）的利用率和可用性。这种意识将有助于了解资源使用情况并有助于控制成本。

## 使用 CloudWatch 容器见解

使用 CloudWatch Container Insights 收集、汇总和汇总来自容器化应用程序和微服务的指标和日志。容器见解适用于亚马逊 Elastic Kubernetes Service、亚马逊上的 Kubernetes 平台和亚马逊 EC2上的 Kubernetes 平台。 EC2指标包括资源的使用率，如 CPU、内存、磁盘和网络。

文档中给出了见解的安装。

CloudWatch 创建集群、节点、Pod、任务和服务级别的聚合指标作为 CloudWatch 指标。

以下查询显示了按节点 CPU 平均利用率排序的节点列表

```
STATS avg(node_cpu_utilization) as avg_node_cpu_utilization by NodeName
| SORT avg_node_cpu_utilization DESC
```

按容器名称划分的 CPU 使用率

```
stats pct(container_cpu_usage_total, 50) as CPUPercMedian by kubernetes.container_name
| filter Type="Container"
```

按容器名称划分的磁盘使用情况

```
stats floor(avg(container_filesystem_usage/1024)) as container_filesystem_usage_avg_kb
 by InstanceId, kubernetes.container_name, device
| filter Type="ContainerFS"
| sort container_filesystem_usage_avg_kb desc
```

更多示例查询在《容器见解》文档中给出

这种意识将有助于了解资源使用情况并有助于控制成本。

## 使用 Kubecost 进行支出意识和指导

诸如 kubecost 之类的第三方工具也可以部署在 Amazon EKS 上，以了解运行 Kubernetes 集群的成本。有关使用 Kubecost 跟踪成本的信息，请参阅此 AWS 博客

使用 Helm 3 部署 kubecost：

```
$ curl -sSL https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
 bash
$ helm version --short
v3.2.1+gfe51cd1
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/c^C
$ kubectl create namespace kubecost
namespace/kubecost created
$ helm repo add kubecost https://kubecost.github.io/cost-analyzer/
"kubecost" has been added to your repositories

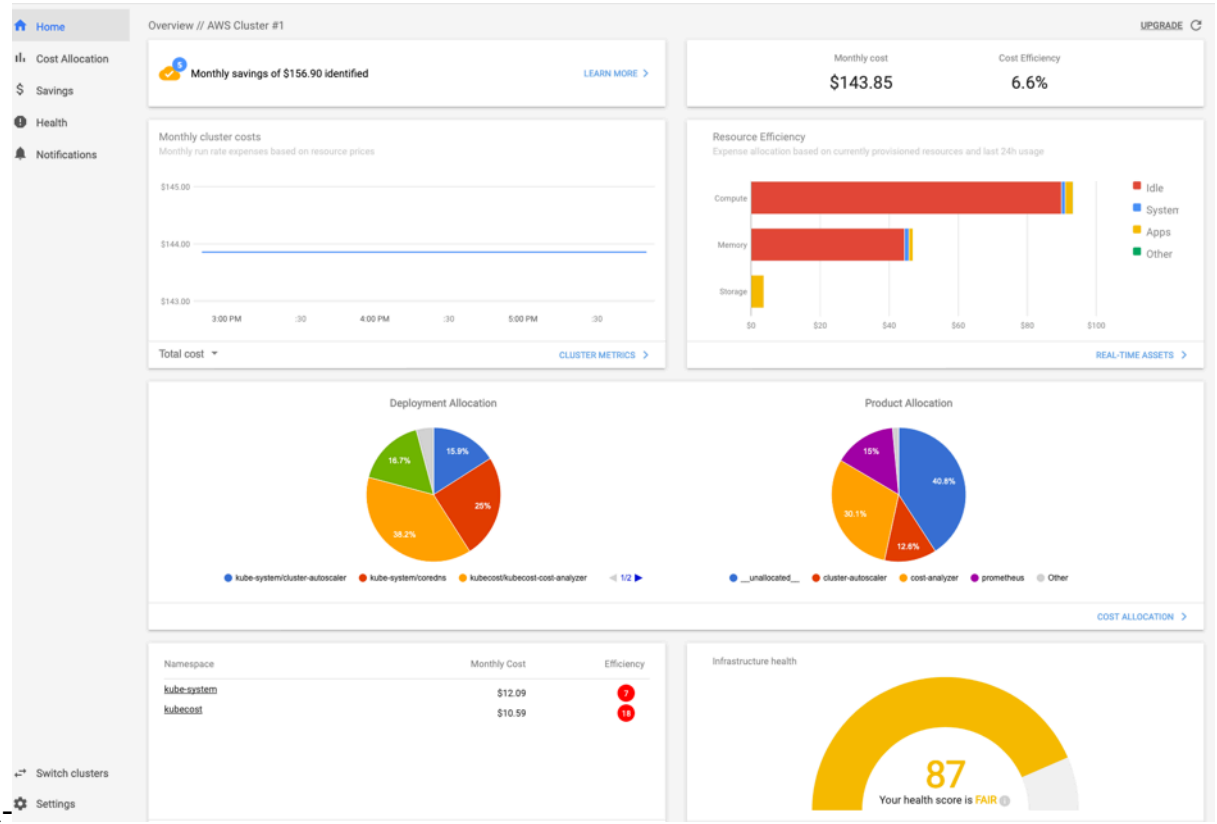
$ helm install kubecost kubecost/cost-analyzer --namespace kubecost --set
 kubecostToken="aGRoZEBqc2pzLmNvbQ==xm343yadf98"
NAME: kubecost
LAST DEPLOYED: Mon May 18 08:49:05 2020
NAMESPACE: kubecost
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
----------------------------------------------------Kubecost has been successfully
 installed. When pods are Ready, you can enable port-forwarding with the following
 command:

    kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

Next, navigate to http://localhost:9090 in a web browser.
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

NOTE: If you are using Cloud 9 or have a need to forward it to a different port like
 8080, issue the following command
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 8080:9090
```

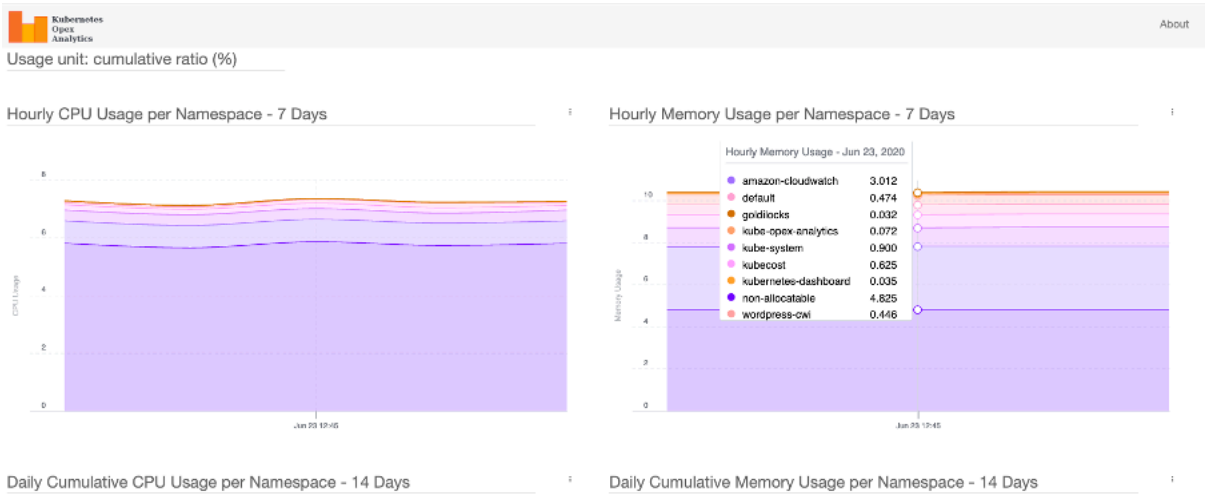
Kubecost 控制面




板-

## 使用 Kubernetes 成本分配和容量规划分析工具

Kubernetes Opex Analytics 是一种工具，可帮助组织跟踪其 Kubernetes 集群消耗的资源，以防止多付钱。为此，它会生成短期（7 天）、中期（14 天）和长期（12 个月）使用报告，显示有关每个项目在一段时间内花费的资源量的相关见解。



### Yotascale

Yotascale 有助于准确分配 Kubernetes 成本。Yotascale Kubernetes 成本分配功能利用实际成本数据（包括预留实例折扣和竞价实例定价，而不是通用的市场价格估计）来告知 Kubernetes 的总成本占用空间

更多细节可以在他们的网站上找到。

### Alcide 顾问

Alcide 是 AWS 合作伙伴网络 (APN) 的高级技术合作伙伴。Alcide Advisor 可帮助确保您的 Amazon EKS 集群、节点和容器配置经过调整，使其根据安全最佳实践和内部指南运行。Alcide Advisor 是一项针对 Kubernetes 审计和合规性的无代理服务，旨在通过在进入生产之前强化开发阶段来确保 DevSecOps 流畅和安全的流程。

更多细节可以在此博客文章中找到。

## 其他工具

### Kubernetes 垃圾收集

Kubernetes 垃圾收集器的作用是删除某些曾经拥有所有者但不再拥有所有者的对象。

### Fargate 计数

Fargatecount 是一个有用的工具，它允许 AWS 客户使用自定义 CloudWatch 指标跟踪特定账户特定区域中已在 Fargate 上部署的 EKS 容器总数。这有助于跟踪在 EKS 集群中运行的所有 Fargate 吊舱。

### Popeye-Kubernetes 集群消毒器

Popeye-Kubernetes 集群清理器是一个实用程序，用于扫描实时的 Kubernetes 集群并报告已部署资源和配置的潜在问题。它会根据已部署的内容而不是磁盘上的内容对集群进行清理。通过扫描您的集群，它可以检测错误配置并帮助您确保最佳实践到位

### 资源

要详细了解成本优化的最佳实践，请参阅以下资源。

文档和博客

- 亚马逊 EKS 支持标记

工具

- 什么是 AWS 账单和成本管理？
- Amazon CloudWatch 容器洞察
- 如何使用 Kubecost 跟踪多租户 Amazon EKS 集群的成本
- Kubecost
- Kube Opsview
- Kubernetes 运营支出分析

# 计算和自动缩放

作为开发人员，您需要估算应用程序的资源需求，例如CPU和内存，但是如果您不不断调整它们，它们可能会过时，这可能会增加您的成本并降低性能和可靠性。持续调整应用程序的资源需求比第一次就正确调整它们更为重要。

下面提到的最佳实践将帮助您构建和运营成本感知型工作负载，从而实现业务成果，同时最大限度地降低成本，并使您的组织能够最大限度地提高投资回报。优化集群计算成本的高级重要性顺序是：

1. 适当调整工作负载规模
2. 减少未使用的容量
3. 优化计算容量类型（例如 Spot）和加速器（例如 GPUs）

## 正确调整工作负载规模

在大多数 EKS 集群中，大部分成本来自用于运行容器化工作负载的 EC2 实例。如果不了解自己的工作负载需求，您将无法调整计算资源的大小。因此，您必须使用适当的请求和限制，并根据需要对这些设置进行调整。此外，实例大小和存储选择等依赖关系可能会影响工作负载性能，从而对成本和可靠性产生各种意想不到的后果。

请求应与实际利用率保持一致。如果容器的请求过高，就会出现未使用的容量，这是集群总成本的一个重要因素。Pod 中的每个容器（例如应用程序和 sidecar）都应设置自己的请求和限制，以确保集合 pod 限制尽可能准确。

使用 Goldilocks、KRR 和 K ubecost 等工具来估算容器的资源请求和限制。根据应用程序的性质、performance/cost 要求和复杂性，您需要评估哪些指标最适合扩展，应用程序性能在什么时候下降（饱和点），以及如何相应地调整请求和限制。有关此主题的进一步指导，请参阅应用程序的正确尺寸。

我们建议使用水平容器自动扩缩器 (HPA) 来控制应运行应用程序的副本数量，使用垂直容器自动扩缩器 (VPA) 来调整应用程序每个副本所需的请求数量和限制，使用像 Karpent er 或 Cluster Autoscaler 这样的节点自动扩缩器来持续调整集群中的节点总数。本文档的后面部分将记录使用 Karpenter 和 Cluster Autoscaler 的成本优化技术。

Vertical Pod Autoscaler 可以调整分配给容器的请求和限制，从而使工作负载以最佳方式运行。你应该在审计模式下运行 VPA，这样它就不会自动进行更改和重启你的 pod。它将根据观察到的指标提出更改建议。对于影响生产工作负载的任何更改，您都应先在非生产环境中查看和测试这些更改，因为这些更改可能会影响应用程序的可靠性和性能。

## 减少消费

节省资金的最佳方法是配置更少的资源。实现这一目标的一种方法是根据工作负载的当前需求调整工作负载。在开始任何成本优化工作时，您都应确保您的工作负载定义其要求并动态扩展。这将需要从您的应用程序中获取指标并设置配置（例如PodDisruptionBudgets和 Pod Readiness Gates），以确保您的应用程序可以安全地动态地向上和向下扩展。重要的是要考虑到，限制 PodDisruptionBudgets 会阻止 Cluster Autoscaler 和 Karpenter 缩小节点规模，因为集群自动扩缩器和 Karpenter 都尊重。PodDisruptionBudgets中的 “minAvailable” 值 PodDisruptionBudget 应始终低于部署中的容器数量，并且应在两者之间保持良好的缓冲区。例如，在 6 个 pod 的部署中，您希望始终至少运行 4 个 pod，请将您 PodDisruptionBidget 的 “minAvailable” 设置为 4。这将允许 Cluster Autoscaler 和 Karpenter 在节点缩减事件期间安全地从未充分利用的节点中排出 Pod 并将其驱逐出去。请参阅集群自动扩缩器常见问题解答文档。

Horistant Pod Autoscaler 是一款灵活的工作负载自动扩缩器，可以调整需要多少副本来满足应用程序的性能和可靠性要求。它有一个灵活的模型，可以根据各种指标（例如 CPU、内存或自定义指标（例如队列深度、与 Pod 的连接数等）来定义何时向上和向下扩展。

Kubernetes Metrics Server 支持根据内置指标（例如 CPU 和内存使用情况）进行扩展，但是如果您想根据其他指标（例如 Amazon CloudWatch 或 SQS 队列深度）进行扩展，则应考虑事件驱动的自动扩展项目，例如 KEDA。请参阅此博客文章，了解如何将 KEDA 与 CloudWatch 指标配合使用。如果您不确定要根据哪些指标进行监控和扩展，请查看有关重要监控指标的最佳实践。

减少工作负载消耗会在集群中造成容量过剩，通过适当的自动扩展配置，您可以自动缩减节点并减少总支出。我们建议您不要尝试手动优化计算容量。Kubernetes 调度器和节点自动扩缩器旨在为您处理此过程。

# 减少未使用的容量

在确定了应用程序的正确大小并减少了多余的请求之后，就可以开始减少预配置的计算容量了。如果您已经花时间根据上面的部分正确调整工作负载的大小，则应该能够动态地执行此操作。AWS 中有两个主节点自动扩缩器与 Kubernetes 配合使用。

## Karpenter 和 Cluster 自动扩缩器

随着 Pod 的创建或移除以及计算要求的变化，Karpenter 和 Kubernetes 集群自动扩缩器都会扩展集群中的节点数量。两者的主要目标是一样的，但是 Karpenter 在节点管理配置和取消配置方面采用了不同的方法，这有助于降低成本并优化集群范围的使用率。

随着集群规模的扩大和工作负载种类的增加，预配置节点组和实例变得越来越困难。就像处理工作负载请求一样，设定初始基准并根据需要不断调整非常重要。

如果您使用的是集群自动扩缩器，它将尊重每个 Auto Scaling 组 (ASG) 的 “最小” 和 “最大” 值，并且只调整 “所需值”。在为底层 ASG 设置这些值时务必注意，因为 Cluster Autoscaler 无法将超出 “最小” 数量的 ASG 缩小规模。将 “期望” 计数设置为正常工作时间所需的节点数，将 “最小” 设置为非工作时间所需的节点数。请参阅集群自动扩缩器常见问题解答文档。

## 集群自动扩缩器优先级扩展器

Kubernetes Cluster Autoscaler 的工作原理是在应用程序向上和向下扩展时向上和向下扩展节点组（称为节点组）。如果您没有动态扩展工作负载，那么集群自动扩缩器将无法帮助您节省资金。Cluster Autoscaler 要求集群管理员提前创建节点组以供工作负载使用。节点组需要配置为使用具有相同 “配置文件” 的实例，即大致相同的 CPU 和内存量。

您可以有多个节点组，并且可以将 Cluster Autoscaler 配置为设置优先级扩展级别，并且每个节点组可以包含不同大小的节点。节点组可以有不同的容量类型，优先级扩展器可以先扩展成本较低的组。

以下是集群配置片段的示例，该片段在使用按需实例之前使用`ConfigMap`来确定预留容量的优先级。您可以使用相同的方法将 Graviton 或 Spot 实例优先于其他类型。

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-cluster
managedNodeGroups:
  - name: managed-ondemand
    minSize: 1
    maxSize: 7
    instanceType: m5.xlarge
```

```
  - name: managed-reserved
    minSize: 2
    maxSize: 10
    instanceType: c5.2xlarge
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*ondemand.*
    50:
      - .*reserved.*
```

默认情况下，使用节点组可以帮助底层计算资源完成预期的事情，例如将节点分布在各处 AZs，但并非所有工作负载都有相同的要求或期望，最好让应用程序明确声明其需求。有关集群自动扩缩器的更多信息，请参阅最佳实践部分。

## Descheduler

Cluster Autoscaler 可以根据需要调度的新 Pod 或未充分利用的节点来增加和移除集群中的节点容量。在将 pod 调度到节点后，它不会全面了解 pod 的放置情况。如果您使用的是集群自动扩缩程序，则还应查看 Kubernetes 解调器，以避免浪费集群中的容量。

如果您的集群中有 10 个节点，并且每个节点的利用率为 60%，则您使用的容量不会占集群中预置容量的 40%。使用 Cluster Autoscaler，您可以将每个节点的利用率阈值设置为 60%，但只有在利用率降至 60% 以下之后，才会尝试缩小单个节点。

使用解调器，它可以在调度 pod 或将节点添加到集群后查看集群容量和利用率。它会尝试将集群的总容量保持在指定的阈值以上。它还可以根据节点污点或加入集群的新节点移除 Pod，以确保 Pod 在其最佳计算环境中运行。请注意，descheduler 不会安排更换已驱逐的 pod，而是依赖默认的调度器来完成此操作。

## Karpenter 整合

Karpenter 采用 “无组” 方法进行节点管理。这种方法对于不同的工作负载类型更加灵活，并且需要更少的集群管理员预先配置。Karpenter 没有预先定义组并根据工作负载的需要扩展每个组，而是使用配置器和节点模板来广泛定义可以创建的 EC2 实例类型以及创建实例时的设置。

Bin packing 是一种通过将更多工作负载打包到更少、大小最优的实例上来利用更多实例资源的做法。虽然这仅通过配置工作负载使用的资源来帮助降低计算成本，但它需要权衡取舍。启动新工作负载可能需要更长时间，因为必须向集群添加容量，尤其是在大规模扩展活动期间。设置垃圾箱包装时，请考虑成本优化、性能和可用性之间的平衡。

Karpenter 可以持续监控和装箱，以提高实例资源利用率并降低计算成本。Karpenter 还可以为您的工作负载选择更具成本效益的工作节点。这可以通过在配置器中将 “合并” 标志设置为 true 来实现（下面的示例代码片段）。以下示例显示了一个支持整合的置备程序示例。在撰写本指南时，Karpenter 不会将正在运行的竞价型实例替换为更便宜的竞价型实例。有关 Karpenter 整合的更多详细信息，请参阅此博客。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: enable-binpacking
spec:
  consolidation:
    enabled: true
```

对于可能无法中断的工作负载，例如没有检查点的长时间运行的批处理作业，可以考虑使用注释为 pod 添加注释。do-not-evict通过选择 pod 退出驱逐，你是在告诉 Karpenter 它不应该自愿移除包含这个 pod 的节点。但是，如果在节点耗尽时将 do-not-evict Pod 添加到该节点，则其余的 pod 仍会被驱逐，但是该 Pod 在被移除之前会阻止终止。无论哪种情况，都将封锁该节点，以防止在该节点上安排其他工作。以下是显示如何设置注释的示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "karpenter.sh/do-not-evict": "true"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

## 通过调整集群自动扩缩程序参数来移除未充分利用的节点

节点利用率定义为请求的资源总和除以容量。默认情况下`scale-down-utilization-threshold`，设置为 50%。此参数可以与和一起使用`scale-down-unneeded-time`，后者决定了节点在有资格缩减之前应该不需要多长时间，默认值为 10 分钟。Kube-scheduler 会将仍在缩减的节点上运行的 Pod 调度到其他节点上。调整这些设置可以帮助移除未充分利用的节点，但请务必先测试这些值，这样就不会强迫集群过早缩小规模。

您可以确保驱逐成本高昂的 Pod 受到集群自动扩缩器识别的标签的保护，从而防止缩小规模的发生。为此，请确保驱逐成本高昂的 pod 具有注释`cluster-autoscaler.kubernetes.io/safe-to-evict=false`。以下是设置注释的 yaml 示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "cluster-autoscaler.kubernetes.io/safe-to-evict": "false"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

## 使用集群自动扩缩器和 Karpenter 标记节点

AWS 资源标签用于组织您的资源，并详细跟踪您的 AWS 成本。它们不与 Kubernetes 标签直接关联以进行成本跟踪。建议从 Kubernetes 资源标签开始，然后使用 Kubecos t 之类的工具根据容器、命名空间等上的 K ubernetes 标签来获取基础设施成本报告。

工作节点需要有标签才能在 AWS Cost Explorer 中显示账单信息。使用 Cluster Autoscaler，使用启动模板在托管节点组中标记您的工作节点。对于自我管理的节点组，请使用 a EC2 uto Scaling 组标记您的实例。对于 Karpenter 配置的实例，请在节点模板中使用 spec.tags对其进行标记。

### 多租户集群

在处理由不同团队共享的集群时，您可能无法看到在同一节点上运行的其他工作负载。虽然资源请求可以帮助隔离一些“邻居噪音大”的问题，例如 CPU 共享，但它们可能无法隔离所有资源边界，例如磁盘 I/O 耗尽。并非每个工作负载消耗的资源都可以被隔离或限制。应通过节点污点和容忍度来隔离以比其他工作负载更高的速率消耗共享资源的工作负载。处理此类工作负载的另一种高级技术是 CPU 固定，它可确保容器专用 CPU 而不是共享 CPU。

在节点级别隔离工作负载可能会更昂贵，但可以通过使用预留实例、Graviton 处理器或 Spot 来安排BestEffort任务或利用额外节省的费用。

共享集群还可能存在集群级别的资源限制，例如 IP 耗尽、Kubernetes 服务限制或 API 扩展请求。您应查看可扩展性最佳实践指南，确保您的集群规避这些限制。

您可以在命名空间或 Karpenter 配置器级别隔离资源。资源配额提供了一种设置命名空间中工作负载可以消耗的资源数量限制的方法。这可能是一个不错的初始防护栏，但应不断对其进行评估，以确保它不会人为地限制工作负载的扩展。

Karpenter Provisioners 可以对集群中的某些消耗资源（例如 CPU、GPU）设置限制，但您需要配置租户应用程序才能使用相应的配置器。这可以防止单个配置器在集群中创建太多节点，但应持续对其进行评估，以确保限制设置得不太低，从而防止工作负载扩展。

### 定时自动缩放

您可能需要在周末和非工作时间缩小集群。这对于测试和非生产集群尤其重要，在这些集群中，您希望在不使用时将其缩小到零。诸如集群关闭之类的解决方案可以根据cron计划将副本缩小到零。您也可以使用 Karpenter 实现这一目标，下面的 AWS 博客对此进行了概述。

## 优化计算容量类型

在优化了集群中的总计算容量并利用垃圾箱打包后，您应该查看在集群中预配置了哪种类型的计算以及如何为这些资源付费。AWS 的计算节省计划可以降低您的计算成本，我们会将其归类为以下容量类型：

- Spot
- 节省计划
- 按需型
- Fargate

每种容量类型在管理开销、可用性和长期承诺方面都有不同的权衡，您需要决定哪种容量类型适合您的环境。任何环境都不应依赖单一容量类型，您可以在单个集群中混合使用多种运行类型，以优化特定的工作负载要求和成本。

## Spot

Spot 容量类型利用可用区的备用容量配置 EC2 实例。Spot 提供的折扣最大，最高可达 90%，但是当其他地方需要这些实例时，这些实例可能会被中断。此外，可能并不总是有容量来配置新的竞价型实例，并且只要发出中断通知 2 分钟，就可以回收现有的竞价型实例。如果您的应用程序启动或关闭过程很长，则竞价型实例可能不是最佳选择。

竞价计算应使用各种各样的实例类型，以降低竞价容量不可用的可能性。需要处理实例中断才能安全关闭节点。使用 Karpenter 或托管节点组的一部分配置的节点会自动支持实例中断通知。如果您使用的是自管理节点，则需要单独运行节点终止处理程序以优雅地关闭竞价型实例。

可以在单个集群中平衡竞价实例和按需实例。借助 Karpenter，您可以创建加权配置器以实现不同容量类型的平衡。借助 Cluster Autoscaler，您可以创建包含竞价型实例和按需实例或预留实例的混合节点组。

以下是使用 Karpenter 将竞价型实例优先于按需实例的优先级的示例。创建置备程序时，您可以指定竞价型、按需型或两者兼而有之（如下所示）。当您同时指定两者时，如果 pod 没有明确指定需要使用 Spot 还是按需，则在使用分配策略配置节点时，Karpenter 会优先考虑 Spot。price-capacity-optimization

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: spot-prioritized
spec:
  requirements:
    - key: "karpenter.sh/capacity-type"
      operator: In
        values: ["spot", "on-demand"]
```

## Savings Plans、预留实例和 AWS EDP

您可以使用计算节省计划来减少计算支出。节省计划为 1 年或 3 年的计算使用承诺提供更低的价格。该用法可以适用于 EKS 集群中的 EC2 实例，但也适用于任何计算用量，例如 Lambda 和 Fargate。通过节省计划，您可以降低成本，并且仍然可以在承诺期内选择任何 EC2 实例类型。

计算节省计划可以将 EC2 成本降低多达 66%，而无需承诺要使用的实例类型、系列或区域。在您使用实例时，节省的费用会自动应用于实例。

EC2 Instance Savings Plans 承诺使用特定地区和 EC2 系列（例如 C 系列的实例），可节省高达 72% 的计算费用。您可以将使用量转移到该区域内的任何可用区，使用任何一代的实例系列，例如 c5 或 c6，并使用该系列中任意大小的实例。折扣将自动应用于您账户中符合储蓄计划标准的任何实例。

预留实例与 Instance EC2 Savings Plan 类似，但与按需实例相比，它们还可以保证可用区或区域的容量，并可降低高达 72% 的成本。计算出需要多少预留容量后，您可以选择要保留多长时间（1 年或 3 年）。当您在账户中运行这些 EC2 实例时，折扣将自动生效。

客户还可以选择与 AWS 签订企业协议。企业协议让客户可以选择定制最适合其需求的协议。客户可以享受基于 AWS EDP（企业折扣计划）的定价折扣。有关企业协议的更多信息，请联系您的 AWS 销售代表。

### 按需型

与现货相比，按需 EC2 实例具有无中断可用性的优势，而且与储蓄计划相比，无需长期承诺。如果您想降低集群成本，则应减少按需 EC2 实例的使用量。

优化工作负载要求后，您应该能够计算出集群的最小和最大容量。这个数字可能会随着时间的推移而变化，但很少会下降。考虑对低于最低限额的所有内容使用 Savings Plan，并选择不会影响应用程序可用性的容量。任何其他可能无法持续使用或可用性所必需的东西都可以按需使用。

如本节所述，减少使用量的最佳方法是消耗更少的资源，并最大限度地利用您配置的资源。使用集群自动扩缩器，您可以使用该设置删除未充分利用的节点。`scale-down-utilization-threshold`使用 Karpenter 时，建议启用整合。

要手动识别可用于您的工作负载的 EC2 实例类型，您应该使用 ec2-instance-selec tor，它可以显示每个区域中可用的实例以及与 EKS 兼容的实例。具有 x86 进程架构、4 Gb 内存、2 v CPUs 且在 us-east-1 区域可用的实例的用法示例。

```
ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 \
  -r us-east-1 --service eks
c5.large
c5a.large
c5ad.large
c5d.large
c6a.large
c6i.large
t2.medium
t3.medium
```

```
t3a.medium
```

对于非生产环境，您可以在夜间和周末等未使用的时间自动缩小集群。kubecost 项目集群关闭就是控制器的一个示例，它可以根据设定的时间表自动缩小集群。

### Fargate 计算

Fargate 计算是 EKS 集群的完全托管计算选项。它通过在 Kubernetes 集群中为每个节点调度一个容器来提供容器隔离。它允许您根据工作负载的 CPU 和 RAM 要求调整计算节点的大小，从而严格控制集群中的工作负载使用情况。

Fargate 可以将工作负载扩展到 0.25 vCPU，内存为 0.5 GB，内存为 120 GB，内存可扩展至 16 个 vCPU。容器大小可用的变体数量有限制，您需要了解您的工作负载如何最适合 Fargate 配置。例如，如果您的工作负载需要 1 个 vCPU 和 0.5 GB 内存，那么最小的 Fargate 容器将是 1 个 vCPU 和 2 GB 内存。

尽管 Fargate 有许多好处，例如无需 EC2 实例或操作系统管理，但由于每个部署的 Pod 都作为集群中的单独节点隔离，因此它可能需要比传统 EC2 实例更多的计算容量。这需要对诸如 Kubelet、日志代理以及通常 DaemonSets 要部署到节点的任何东西进行更多的复制。 DaemonSets 在 Fargate 中不支持它们，需要将其转换为 pod “`sidecars” 并与应用程序一起运行。

Fargate 无法从 bin 打包或 CPU 过度配置中受益，因为工作负载的边界是一个不可突发的节点，也不能在工作负载之间共享。Fargate 将为您节省 EC2 实例管理时间，而这本身就是有成本的，但是 CPU 和内存成本可能比其他 EC2 容量类型更昂贵。Fargate pod 可以利用计算节省计划来降低按需成本。

## 优化计算使用率

在计算基础设施上节省资金的另一种方法是为工作负载使用更高效的计算。这可能来自性能更高的通用计算，例如 Graviton处理器，它比x86便宜20%，能效高60%，或者工作负载特定的加速器，例如和。GPUs FPGAs你需要构建可以在 arm 架构上运行的容器，并为你的工作负载设置带有正确加速器的节点。

EKS 能够运行混合架构的集群（例如 amd64 和 arm64），如果您的容器是针对多个架构编译的，则可以通过在配置器中允许这两种架构来利用带有 Karpenter 的 Graviton 处理器。但是，为了保持稳定的性能，建议您将每个工作负载保持在单个计算架构上，并且仅在没有额外容量可用时才使用不同的架构。

Provisioner 可以配置多个架构，工作负载也可以在其工作负载规格中请求特定的架构。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
```

```
metadata:
  name: default
spec:
  requirements:
  - key: "kubernetes.io/arch"
    operator: In
    values: ["arm64", "amd64"]
```

使用 Cluster Autoscaler，你需要为 Graviton 实例创建一个节点组，并对工作负载设置节点容忍度才能利用新的容量。

GPUs 并 FPGAs 可以大大提高工作负载的性能，但是需要优化工作负载才能使用加速器。机器学习和人工智能的许多工作负载类型 GPUs 可用于计算，并且可以使用资源请求将实例添加到集群并安装到工作负载中。

```
spec:
  template:
    spec:
    - containers:
      ...
      resources:
          limits:
            nvidia.com/gpu: "1"
```

某些 GPU 硬件可以在多个工作负载之间共享，因此可以配置和使用单个 GPU。要了解如何配置工作负载 GPU 共享，请参阅虚拟 GPU 设备插件了解更多信息。您也可以参考以下博客：

- 使用 NVIDIA 时间切片和加速实例在 Amazon EKS 上共享 GPU EC2
- 在 Amazon EKS 上使用 NVIDIA 的多实例 GPU（MIG），最大限度地提高 GPU 利用率：让每个 GPU 运行更多容器组（pod）以增强性能

# 成本优化-联网

架构系统以实现高可用性 (HA) 是实现弹性和容错能力的最佳实践。实际上，这意味着将您的工作负载和底层基础设施分散到给定 AWS 区域的多个可用区 (AZs)。确保您的 Amazon EKS 环境具备这些特性将增强系统的整体可靠性。除此之外，您的 EKS 环境也可能由各种结构（即 VPCs）、组件（即）和集成（即 ECR 和其他容器注册表 ELBs）组成。

高可用性系统和其他特定用例组件的组合可以在数据的传输和处理方式中发挥重要作用。这反过来又会影响数据传输和处理所产生的成本。

下面详细介绍的实践将帮助您设计和优化 EKS 环境，从而实现不同领域和用例的成本效益。

# Pod 到 Pod 通信

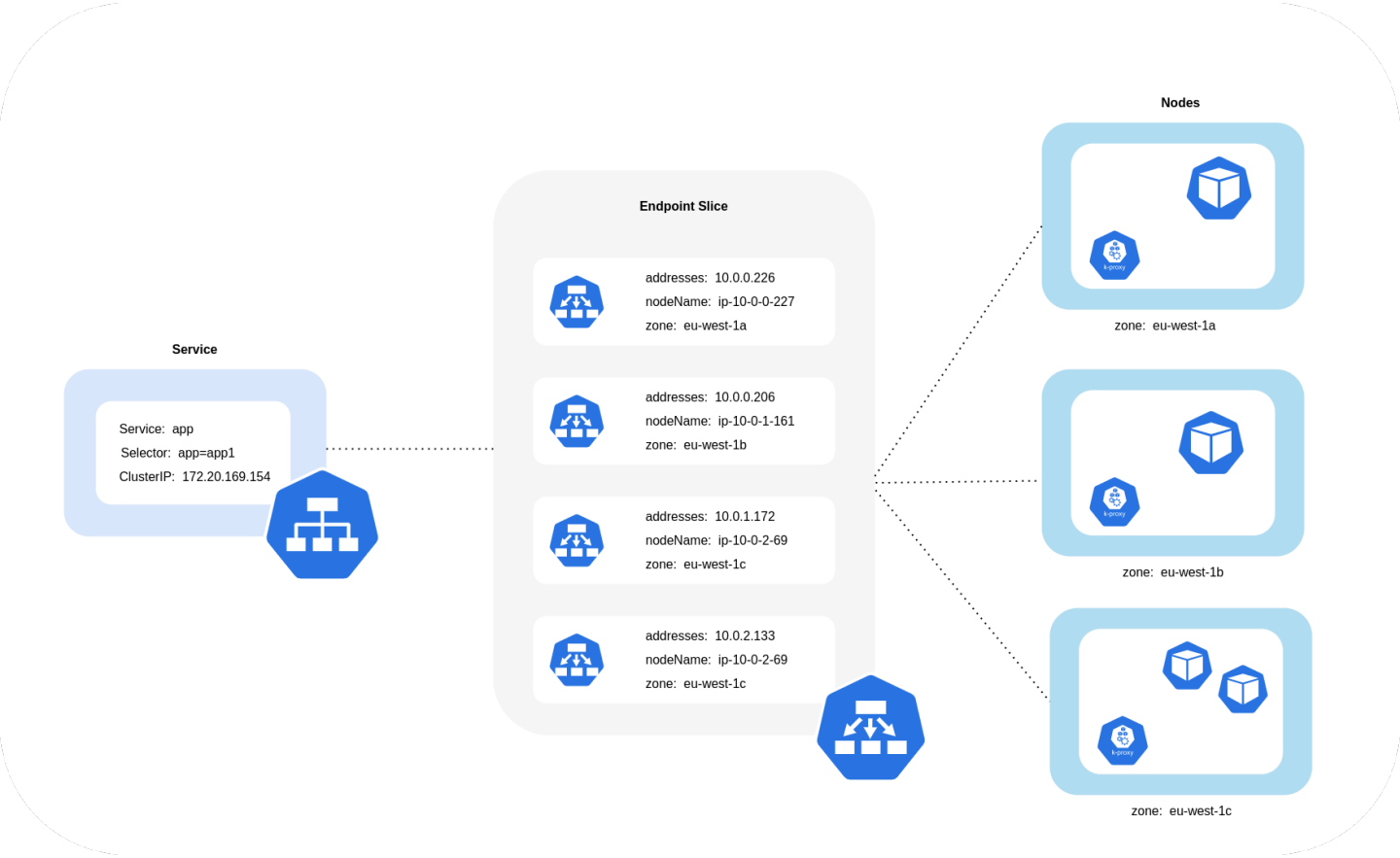
根据您的设置，Pod 之间的网络通信和数据传输可能会对运行 Amazon EKS 工作负载的总体成本产生重大影响。本节将介绍降低与 pod 间通信相关的成本的不同概念和方法，同时考虑高可用性 (HA) 架构、应用程序性能和弹性。

## 限制流向可用区的流量

Kubernetes 项目很早就开始开发拓扑感知结构，包括 kubernetes 等标签。 io/hostname, topology.kubernetes.io/region, and topology.kubernetes.io/zone分配给节点以启用诸如跨故障域分配工作负载和拓扑感知卷配置器之类的功能。在 Kubernetes 1.17 中毕业后，这些标签还被用来为 Pod 到 Pod 的通信启用拓扑感知路由功能。

以下是一些策略，说明如何控制 EKS 集群中 Pod 之间的跨可用区流量以降低成本并最大限度地减少延迟。

如果您想详细了解集群中 Pod 之间的跨可用区流量（例如以字节为单位传输的数据量），请参阅这篇文章。



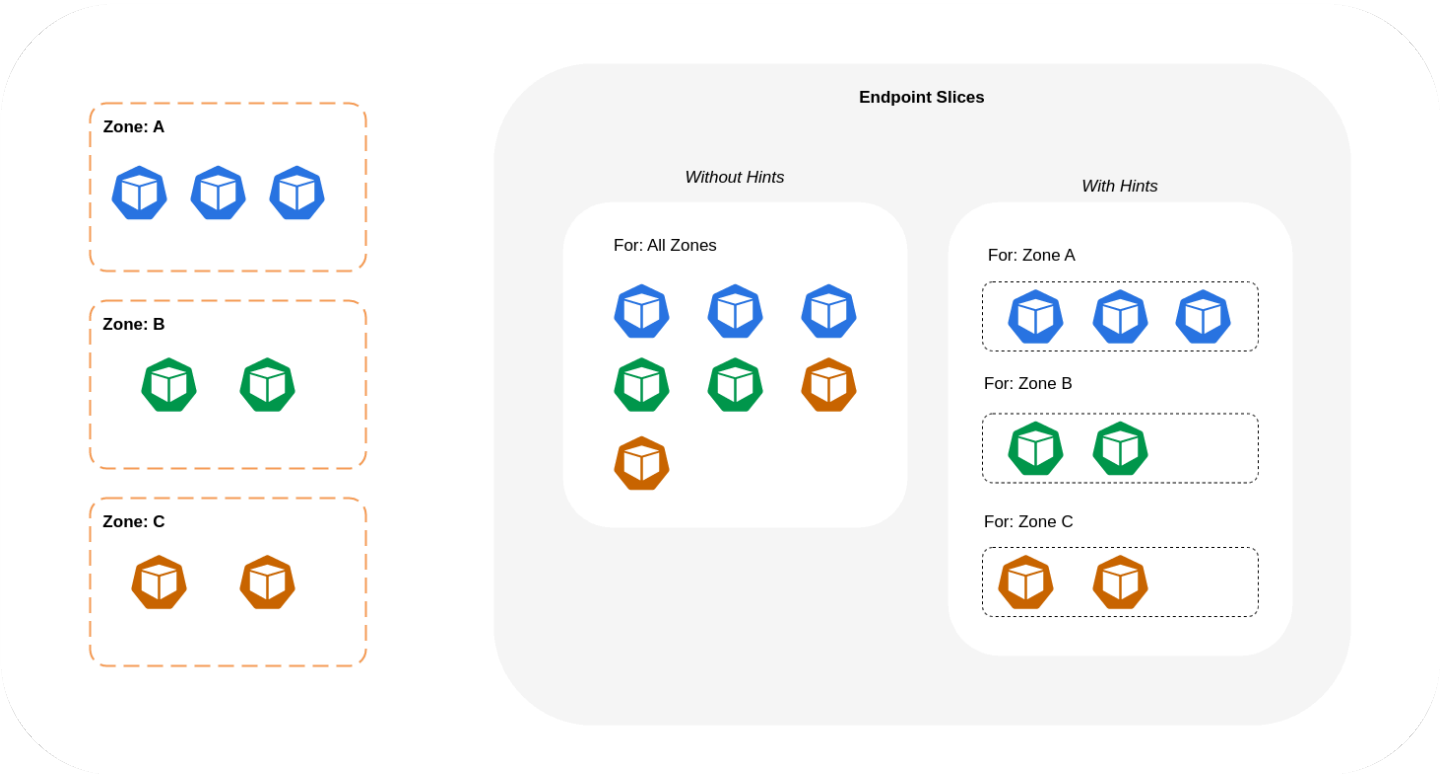
如上图所示，服务是稳定的网络抽象层，用于接收发往您的 Pod 的流量。创建服务时，EndpointSlices 会创建多个服务。每个端点 EndpointSlice 都有一份端点列表，其中包含 Pod 地址的子集，以及它们正在运行的节点和任何其他拓扑信息。在使用 Amazon VPC CNI 时，kube-proxy（在每个节点上运行的守护进程集）会维护网络规则以启用 Pod 通信和服务发现（基于 EBPF 的替代方案可能不使用 kube-p CNIs roxy 但提供等效行为）。它起到了内部路由的作用，但它是根据它从创建的路由中消耗的资源来实现的。 EndpointSlices

在 EKS 上，kube-proxy 主要使用 iptables NAT 规则（或 IPVS NFTables作为替代方案）在集群中的所有 Pod 之间分配流量，无论它们的节点或可用区位置如何。这种默认分布可能导致跨可用区流量路由，从而可能导致敏感应用程序的延迟增加和大型部署中的可用区间数据传输费用。

使用拓扑感知路由（以前称为拓扑感知提示）

在 Kubernetes 服务上启用和实现拓扑感知路由后， EndpointSlice 控制器将按比例将端点分配给集群分布的不同区域。对于每个端点， EndpointSlice 控制器还将为该区域设置一个提示。提示描述了终端节点应为哪个区域提供流量。 `kube-proxy`然后会根据应用的提示将流量从区域路由到终端节点。

下图显示了如何 EndpointSlices 以这样的方式组织提示，即`kube-proxy`可以根据其区域原点知道它们应该去哪个目的地。如果没有提示，就没有这样的分配或组织，无论流量来自哪里，都会被代理到不同的区域目的地。



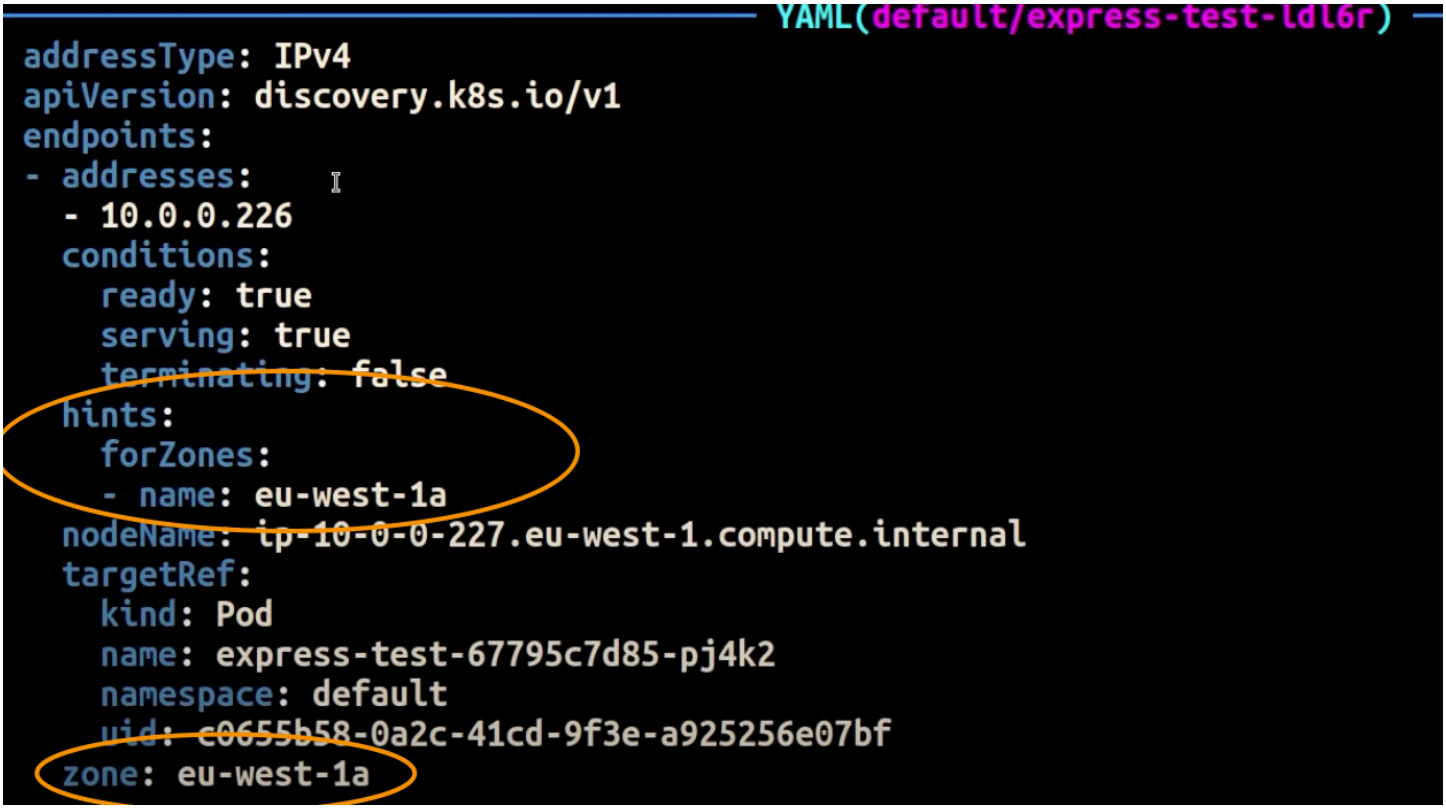
在某些情况下， EndpointSlice 控制器可能会为不同的区域应用提示，这意味着该端点最终可能会为来自不同区域的流量提供服务。这样做的原因是要尝试在不同区域的端点之间保持流量均匀分布。

以下是有关如何为服务启用拓扑感知路由的代码片段。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
  annotations:
    service.kubernetes.io/topology-mode: Auto
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

下面的屏幕截图显示了 EndpointSlice 控制器成功向在 AZ 中运行的 Pod 副本的终端节点应用提示的结果`eu-west-1a`。




> **Note**
>
> 值得注意的是，拓扑感知路由仍处于测试阶段。在整个集群拓扑结构中均匀分布的工作负载时，此功能的性能更具可预测性，因为控制器在各个区域之间按比例分配端点，但是当区域中的节点资源过于不平衡而无法避免过度过载时，可能会跳过提示分配。因此，强烈建议将其与提高应用程序可用性的调度约束（例如 Pod 拓扑分布限制）结合使用。请注意，当容量跨区域波动时（例如使用 Amazon EC2 竞价型实例时），也可能无法分配提示，因为在计算比例分布时不会实时检测到中断或替换。

使用流量分配

流量分布在 Kubernetes 1.30 中引入，在 1.33 中正式推出，它为同区域流量偏好提供了一种更简单的拓扑感知路由替代方案。虽然拓扑感知路由尝试使用智能的流量路由方法来避免端点过载，但它导致了不可预测的行为。相反，流量分配优先考虑可预测性。该 PreferClose 选项指示 kube-proxy 创建规则，根据控制器设置的区域提示，首先将流量路由到同区域端点。 EndpointSlice 当没有同区域终端节点可用时，它会回退为在服务的所有集群终端节点之间分配流量。此功能专为那些接受权衡取舍的工作负载而设计，即优化邻近性，而不是像拓扑感知路由那样尝试均匀分配负载。

以下是有关如何为服务启用流量分配的代码片段。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  trafficDistribution: PreferClose
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

启用流量分布时，会出现一个常见的难题：如果大多数流量来自同一个区域，则单个可用区内的端点可能会过载。这种重载可能会造成重大问题：

- 管理多可用区部署的单个水平 Pod Autoscaler (HPA) 可以通过跨不同区域扩展 pod 来做出响应。AZs但是，此操作无法有效解决受影响区域中增加的负载。
- 这种情况反过来会导致资源效率低下。当像 Karpenter 这样的集群自动扩缩器检测到不同的 pod 横向扩展时 AZs，它们可能会在未受影响的 AZs节点中配置其他节点，从而导致不必要的资源分配。

要克服这一挑战：

- 为每个区域创建单独的部署，这些部署 HPAs 可以相互独立扩展。
- 利用拓扑分布约束来确保工作负载分布在整个集群中，这有助于防止高流量区域中的端点过载。

使用自动扩缩器：将节点配置到特定的可用区

我们强烈建议您在多个高度可用的环境中运行您的工作负载 AZs。这可以提高应用程序的可靠性，尤其是在可用区出现问题时。如果您愿意为了降低网络相关成本而牺牲可靠性，则可以将节点限制为单个可用区。

要在同一个可用区中运行所有 Pod，要么在同一个可用区中配置工作节点，要么将 Pod 调度到在同一个可用区上运行的工作节点上。要在单个可用区内配置节点，请使用集群自动扩缩器 (CA) 定义一个子网属于同一可用区的节点组。对于 Karpenter，请使用`topology.kubernetes.io/zone`并指定要在其中创建工作节点的可用区。例如，下面的 Karpenter 配置程序片段配置了 us-west-2a AZ 中的节点。

Karpenter

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
name: single-az
spec:
  requirements:

* key: "topology.kubernetes.io/zone"`
operator: In
values: ["us-west-2a"]
```

集群自动扩缩器 (CA)

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-ca-cluster
  region: us-east-1
  version: "1.21"
availabilityZones:

* us-east-1a
managedNodeGroups:
* name: managed-nodes
labels:
  role: managed-nodes
instanceType: t3.medium
minSize: 1
maxSize: 10
desiredCapacity: 1
...
```

使用 Pod 分配和节点亲和性

或者，如果您有多个 AZs工作节点运行，则每个节点的标签都将标有 topology.kubernetes.io/zone 以及其可用区的值（例如 us- west-2a 或 us-west-2b）。您可以利用`nodeSelector`或将 Pod 调度`nodeAffinity`到单个可用区中的节点。例如，以下清单文件会将 Pod 调度到在 AZ us-west-2a 中运行的节点内。

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  nodeSelector:
    topology.kubernetes.io/zone: us-west-2a
  containers:

* name: nginx
image: nginx
imagePullPolicy: IfNotPresent
```

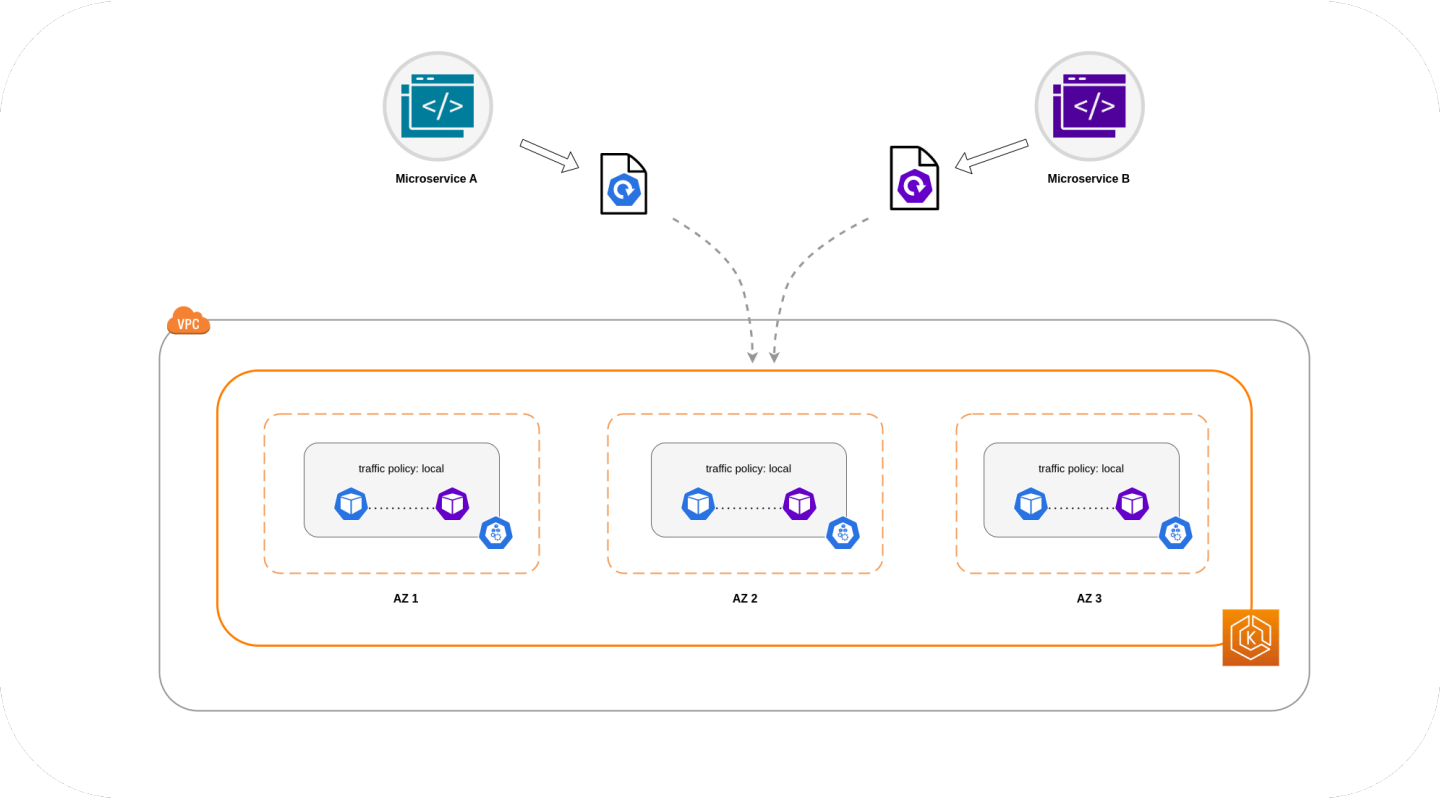
## 限制流向节点的流量

在某些情况下，仅在区域级别限制流量是不够的。除了降低成本外，您可能还需要降低某些经常相互通信的应用程序之间的网络延迟。为了实现最佳网络性能并降低成本，您需要一种方法来限制特定节点的流量。例如，即使在高可用性 (HA) 设置中，微服务 A 也应始终在节点 1 上与微服务 B 通信。让节点 1 上的微服务 A 与节点 2 上的微服务 B 通信可能会对这种性质的应用程序的预期性能产生负面影响，尤其是在节点 2 完全位于单独的可用区时。

### 使用服务内部流量策略

为了限制流向节点的 Pod 网络流量，你可以使用服务内部流量策略。默认情况下，发送到工作负载服务的流量将随机分布在生成的不同端点上。因此，在 HA 架构中，这意味着来自微服务 A 的流量可以流向不同节点上任何给定节点上的微服务 B 的任何副本。 AZs但是，当服务的内部流量策略设置为`Local`时，流量将仅限于流量来源节点上的端点。此政策规定只能使用节点本地端点。这意味着，该工作负载的网络流量相关成本将低于分布在集群范围内的成本。此外，延迟会降低，从而提高应用程序的性能。

> **Note**
>
> 需要注意的是，此功能不能与 Kubernetes 中的拓扑感知路由结合使用。




以下是有关如何为服务设置内部流量策略的代码片段。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

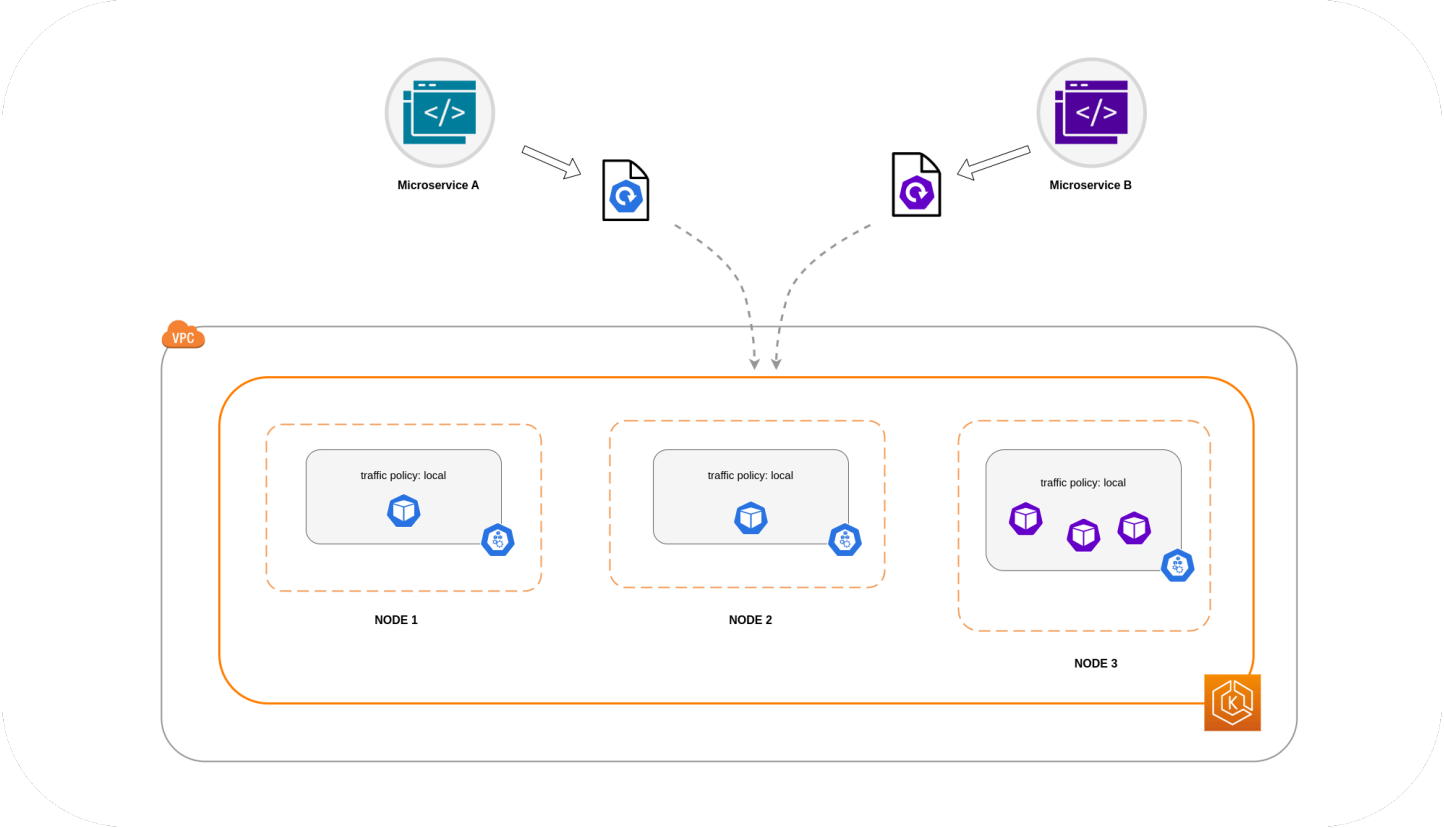
* protocol: TCP
port: 3003
targetPort: 3003
  internalTrafficPolicy: Local
```

为避免应用程序因流量下降而出现意外行为，应考虑以下方法：

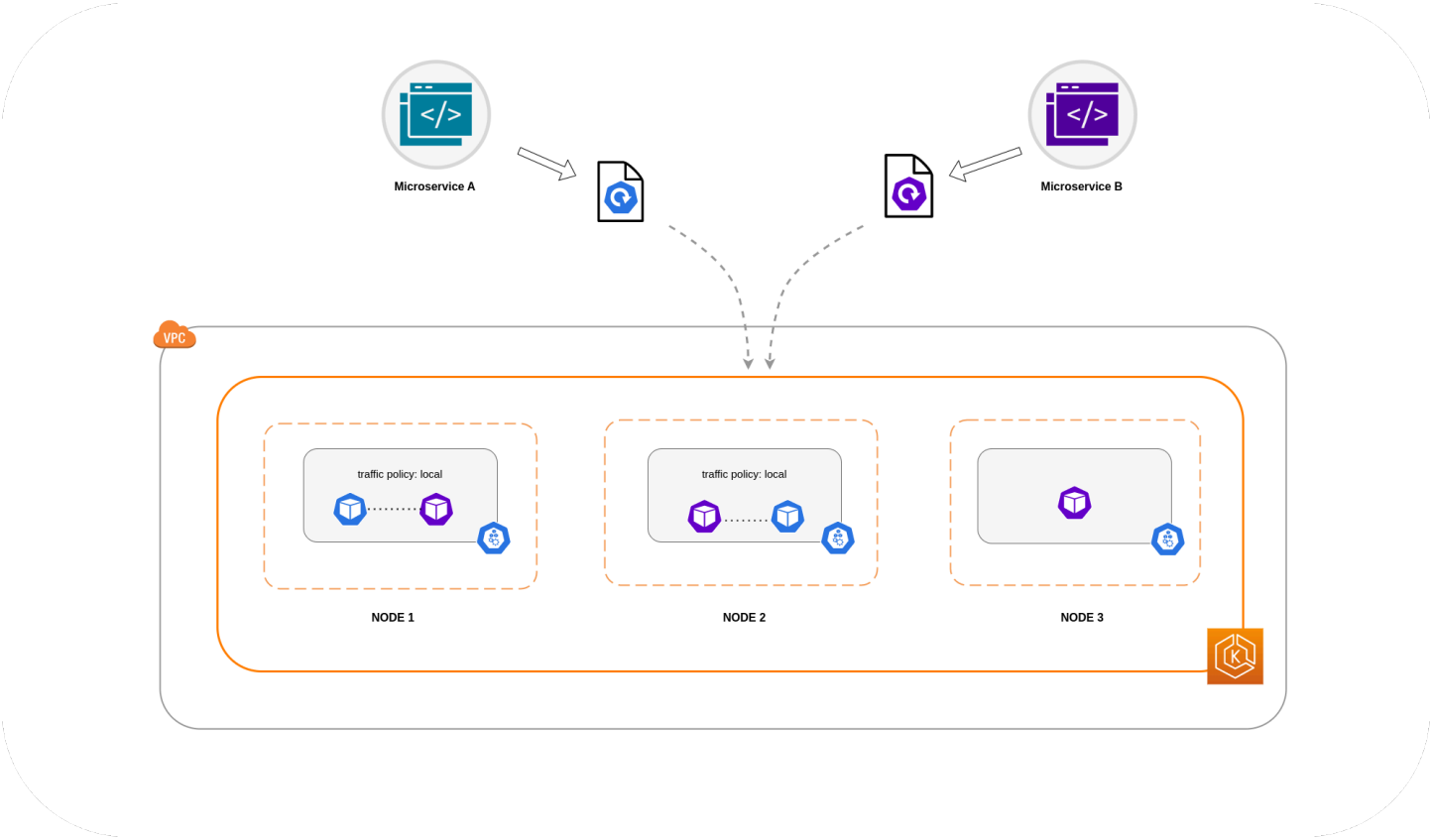
- 为每个通信的 Pod 运行足够的副本

- 使用拓扑分布约束让 Pod 分布相对均匀
- 利用 pod 关联性规则对通信 Pod 进行同地定位

在此示例中，您有 2 个微服务 A 副本和 3 个微服务 B 副本。如果微服务 A 的副本分布在节点 1 和节点 2 之间，而微服务 B 的 3 个副本全部位于节点 3 上，则由于内部流量策略，它们将无法通信。Local当没有可用的节点本地端点时，流量将被丢弃。




如果微服务 B 的 3 个副本中的 2 个位于节点 1 和 2 上，则对等应用程序之间将进行通信。但是你仍然会有一个微服务 B 的隔离副本，没有任何对等副本可以与之通信。




> **Note**
>
> 在某些情况下，像上图所示的隔离副本如果仍有用途（例如处理来自外部传入流量的请求），则可能不会引起担忧。

使用具有拓扑分布约束的服务内部流量策略

将内部流量策略与拓扑分布限制结合使用可能有助于确保您拥有正确数量的副本，以便在不同节点上通信微服务。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: express-test
spec:
  replicas: 6
  selector:
    matchLabels:
      app: express-test
```

```
  template:
    metadata:
      labels:
        app: express-test
        tier: backend
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
```

将服务内部流量策略与 Pod 关联性规则配合使用

另一种方法是在使用服务内部流量策略时使用 Pod 关联性规则。借助 Pod 亲和性，你可以影响调度器将某些 Pod 放在同一个位置，因为它们经常通信。通过对某些 Pod 应用严格的调度约束 (`requiredDuringSchedulingIgnoredDuringExecution`)，当调度器将 Pod 放在节点上时，这将为你提供 Pod 托管的更好结果。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: graphql
  namespace: ecommerce
  labels:
    app.kubernetes.io/version: "0.1.6"
    ...
    spec:
      serviceAccountName: graphql-service-account
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
```

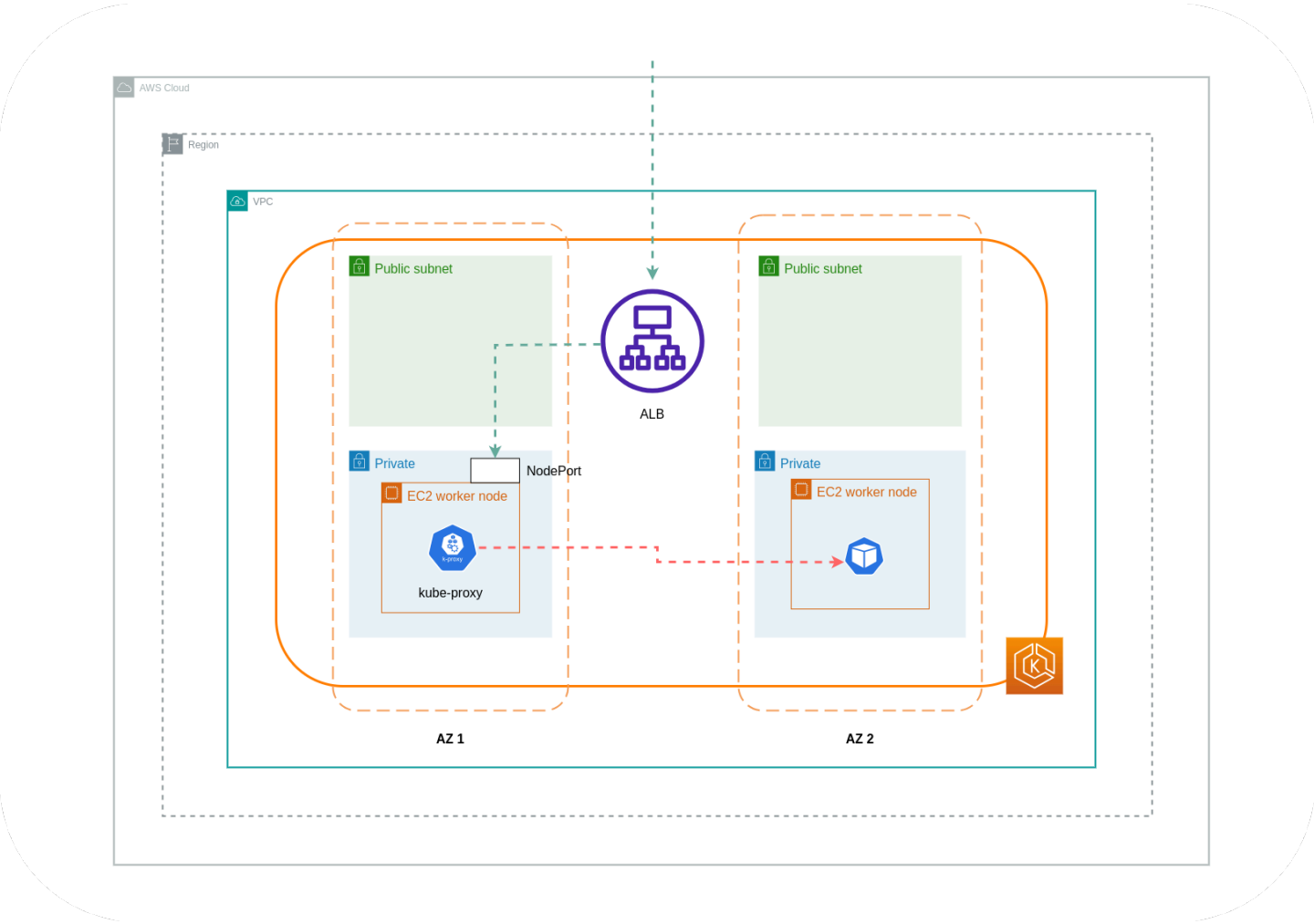
# Load Balancer 到 Pod 通信

EKS 工作负载通常由负载均衡器前置，该负载均衡器将流量分配到 EKS 集群中的相关 Pod。您的架构可能包括面向 and/or 外部的内部负载均衡器。根据您的架构和网络流量配置，负载均衡器与 Pod 之间的通信可能会导致大量的数据传输费用。

您可以使用 AWS Load Balancer 控制器自动管理 ELB 资源（ALB 和 NLB）的创建。在此类设置中产生的数据传输费用将取决于网络流量所采用的路径。AWS Load Balancer 控制器支持两种网络流量模式，即实例模式和 IP 模式。

使用实例模式时， NodePort 将在您的 EKS 集群中的每个节点上打开。然后，负载均衡器将均匀地代理节点间的流量。如果节点上运行了目标 Pod，则不会产生数据传输费用。但是，如果目标 Pod 与 NodePort 接收流量的 Pod 位于不同的节点和不同的可用区中，则从 kube-proxy 到目标 Pod 将有额外的网络跳跃。在这种情况下，将收取跨可用区数据传输费用。由于流量分布均匀，从 kube-proxy 到相关目标 Pod 的跨区域网络流量跳跃很可能会产生额外的数据传输费用。

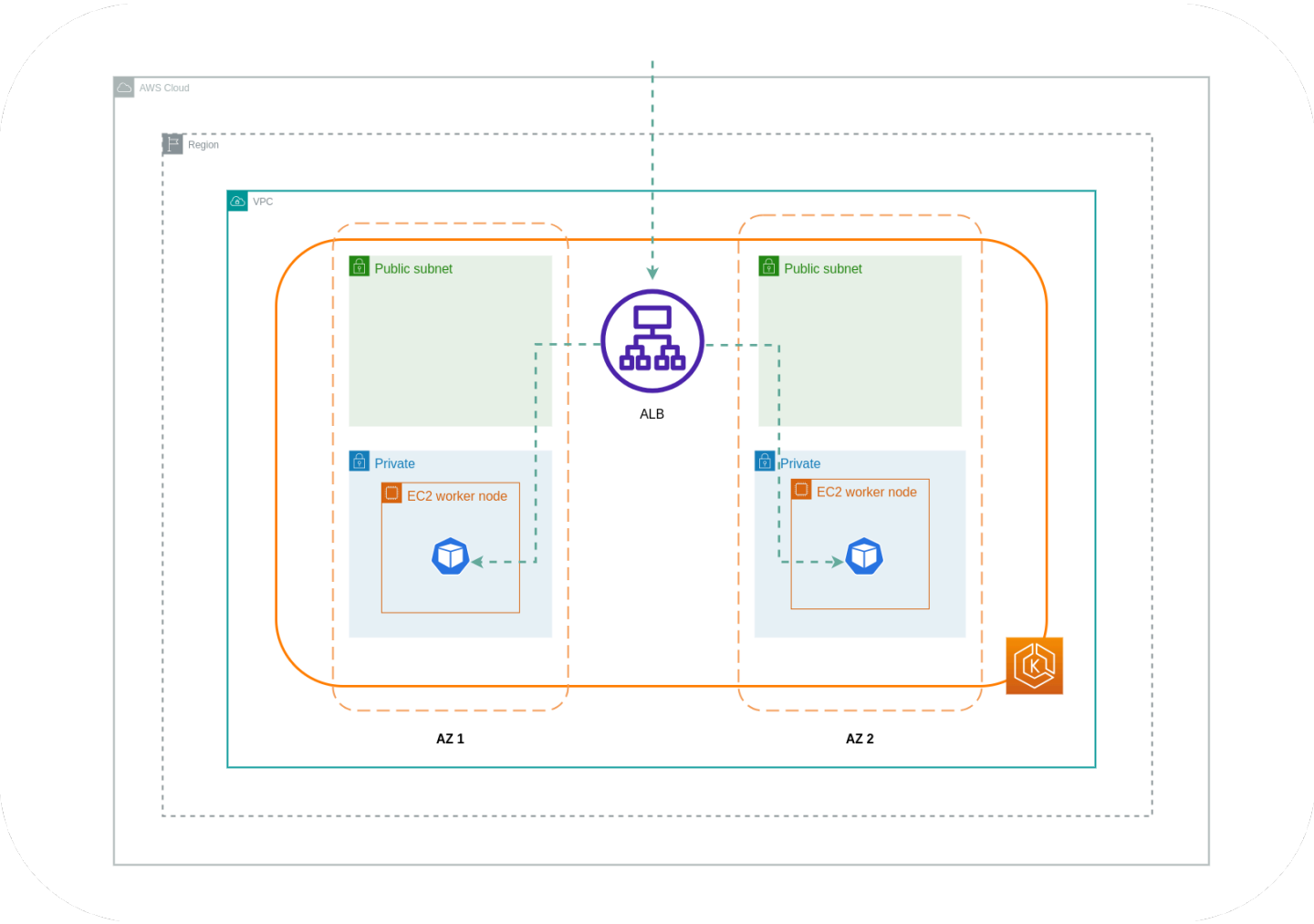
下图描绘了流量从负载均衡器流向另一可用区中单独节点上的目标 Pod NodePort，然后从负载均衡器流`kube-proxy`向目标 Pod 的网络路径。这是实例模式设置的示例。




使用 IP 模式时，网络流量会直接从负载均衡器代理到目标 Pod。因此，这种方法不涉及数据传输费用。

> **Note**
>
> 建议您将负载均衡器设置为 IP 流量模式，以降低数据传输费用。在此设置中，确保您的负载均衡器部署在您的 VPC 中的所有子网中也很重要。

下图描绘了在网络 IP 模式下从负载均衡器流向 Pod 的流量的网络路径。




# 从容器注册表传输数据

## Amazon ECR

向 Amazon ECR 私有注册表传输数据是免费的。区域内数据传输不产生任何费用，但传输到互联网和跨区域的数据传输将按传输双方的互联网数据传输费率收费。

您应该利用 ECRs 内置的映像复制功能将相关的容器镜像复制到与您的工作负载相同的区域。这样，复制只需支付一次费用，并且所有相同的区域（区域内）图像拉取都将是免费的。

通过使用接口 VPC 终端节点连接到区域内 ECR 存储库，可以进一步降低与从 ECR 提取映像（数据传出）相关的数据传输成本。连接到 ECR 的公有 AWS 终端节点（通过 NAT 网关和 Internet Gateway）的替代方法将产生更高的数据处理和传输成本。下一节将更详细地介绍如何降低您的工作负载和 AWS 服务之间的数据传输成本。

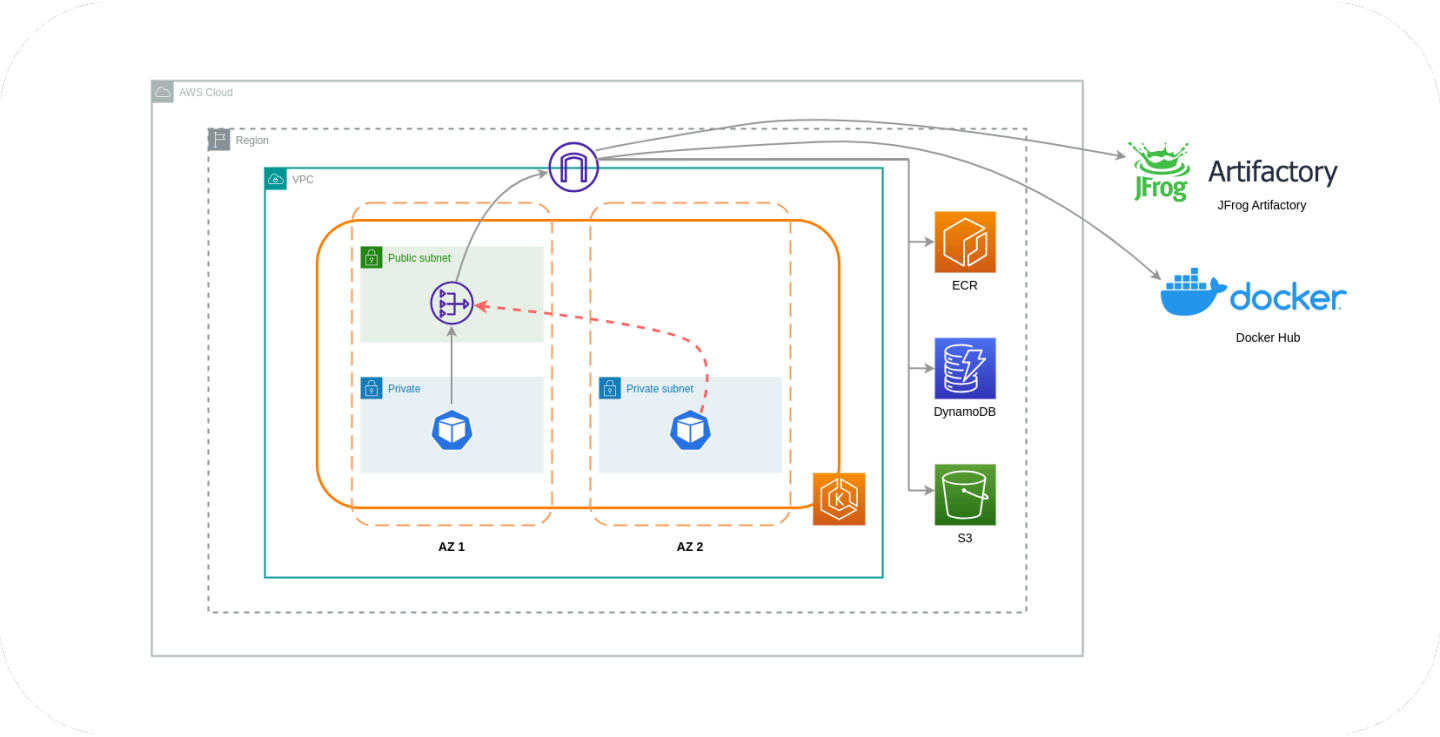
如果您运行的工作负载包含特别大的映像，则可以使用预先缓存的容器映像构建自己的自定义 Amazon 系统映像 (AMIs)。这可以减少初始映像提取时间和从容器注册表到 EKS 工作节点的潜在数据传输成本。

## 将数据传输到互联网和 AWS 服务

通过互联网将 Kubernetes 工作负载与其他 AWS 服务或第三方工具和平台集成，这是一种常见的做法。用于将流量路由到相关目的地的底层网络基础设施可能会影响数据传输过程中产生的成本。

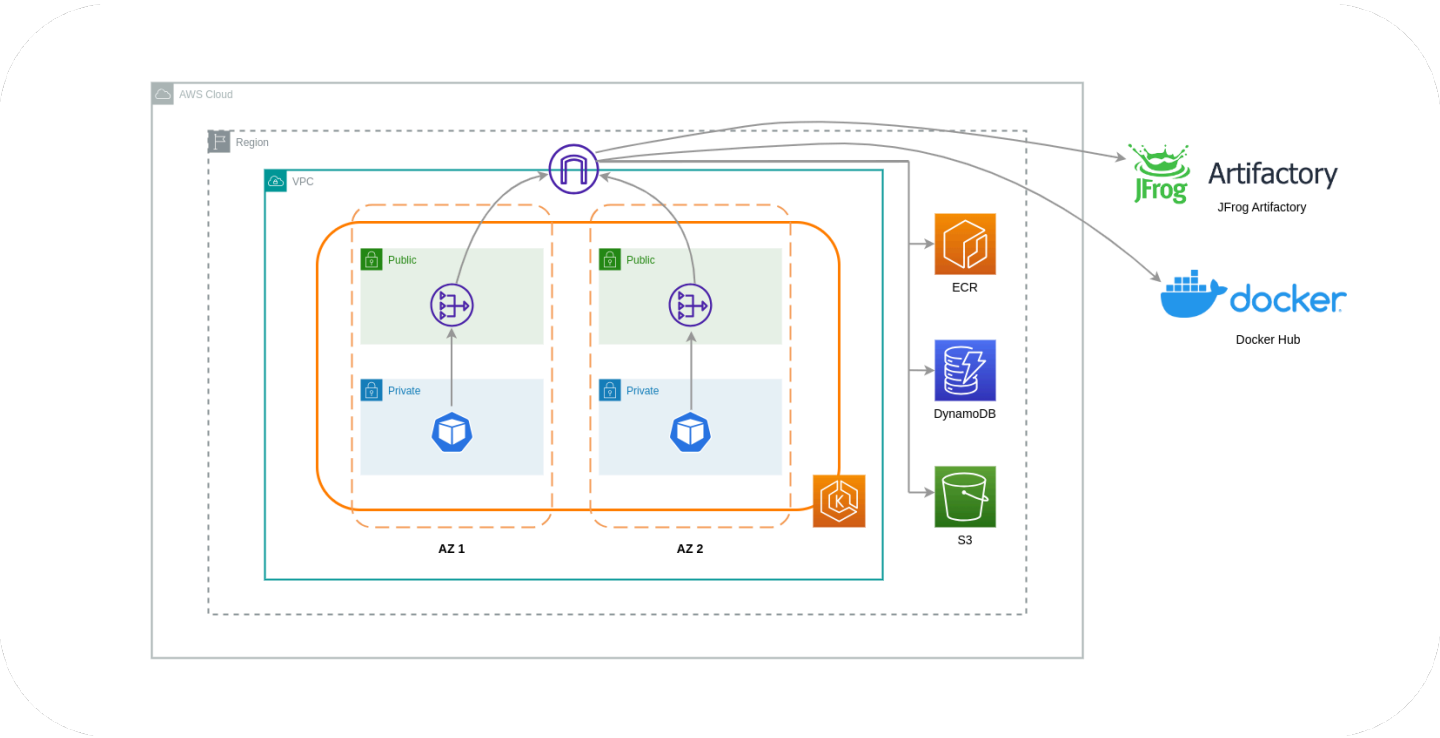
### 使用 NAT 网关

NAT 网关是执行网络地址转换 (NAT) 的网络组件。下图描绘了 EKS 集群中的 Pod 与其他 AWS 服务（亚马逊 ECR、DynamoDB 和 S3）和第三方平台通信。在此示例中，Pod 分别 AZs在私有子网中运行。为了发送和接收来自互联网的流量，需要将 NAT 网关部署到一个可用区的公有子网中，从而允许任何具有私有 IP 地址的资源共享一个公有 IP 地址来访问互联网。反过来，此 NAT 网关与 Internet Gateway 组件通信，允许将数据包发送到其最终目的地。




在此类用例中使用 NAT 网关时，您可以通过在每个可用区部署一个 NAT 网关来最大限度地降低数据传输成本。这样，路由到互联网的流量将通过同一可用区中的 NAT 网关，从而避免可用区间的数据传输。但是，尽管您可以节省可用区间数据传输的成本，但这种设置的含义是，您的架构中会产生额外的 NAT 网关的成本。

这种推荐的方法如下图所示。




## 使用 VPC 终端节点

为了进一步降低此类架构的成本，您应该使用 VPC 终端节点在工作负载和 AWS 服务之间建立连接。VPC 终端节点允许您从 VPC 内部访问 AWS 服务，无需 data/network 数据包通过互联网。所有流量均为内部流量并保持在 AWS 网络内。VPC 终端节点有两种类型：接口 VPC 终端节点（受许多 AWS 服务支持）和网关 VPC 终端节点（仅受 S3 和 DynamoDB 支持）。

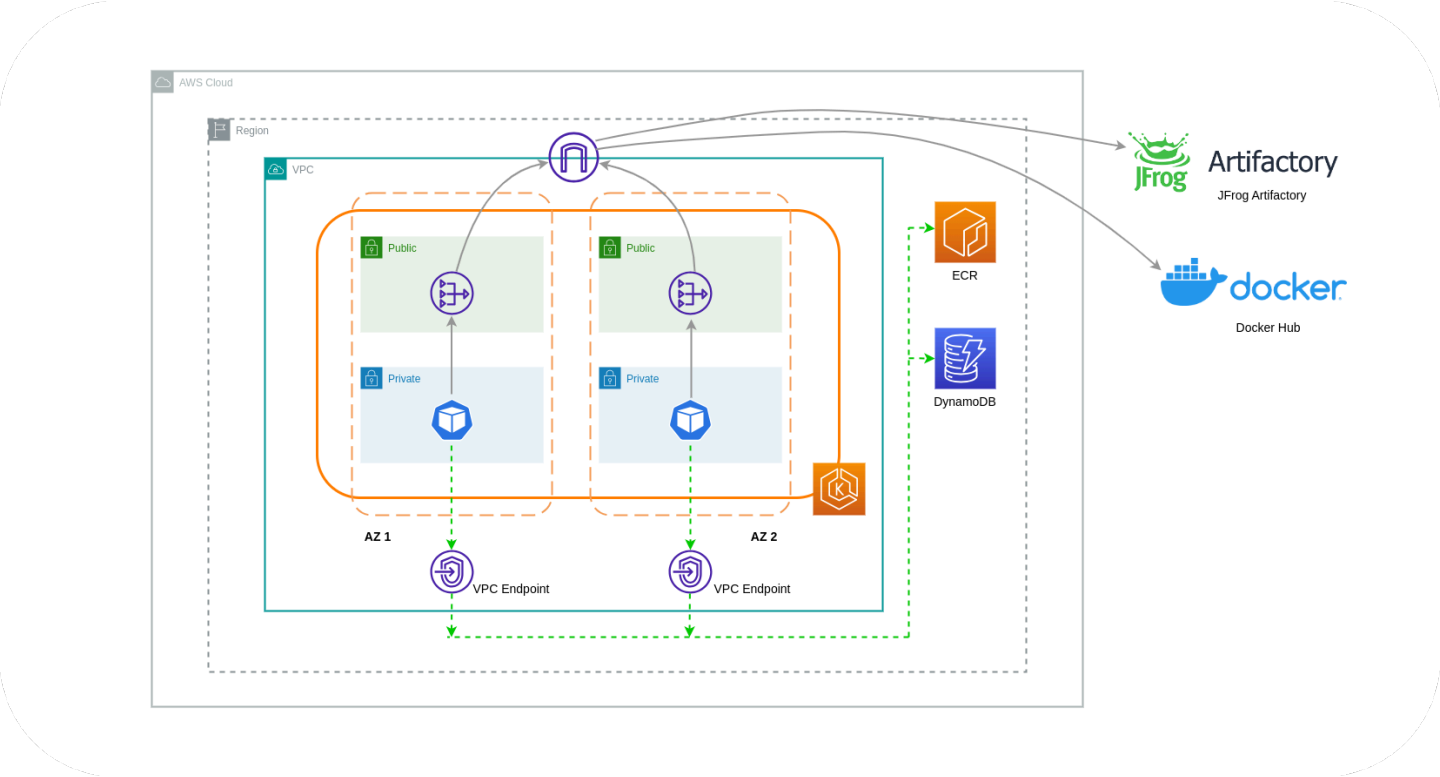
网关 VPC 终端节点

网关 VPC 终端节点不收取每小时费用或数据传输费用。使用网关 VPC 终端节点时，请务必注意，它们不能跨越 VPC 边界扩展。它们不能用于 VPC 对等互连、VPN 网络或通过 Direct Connect。

接口 VPC 终端节点

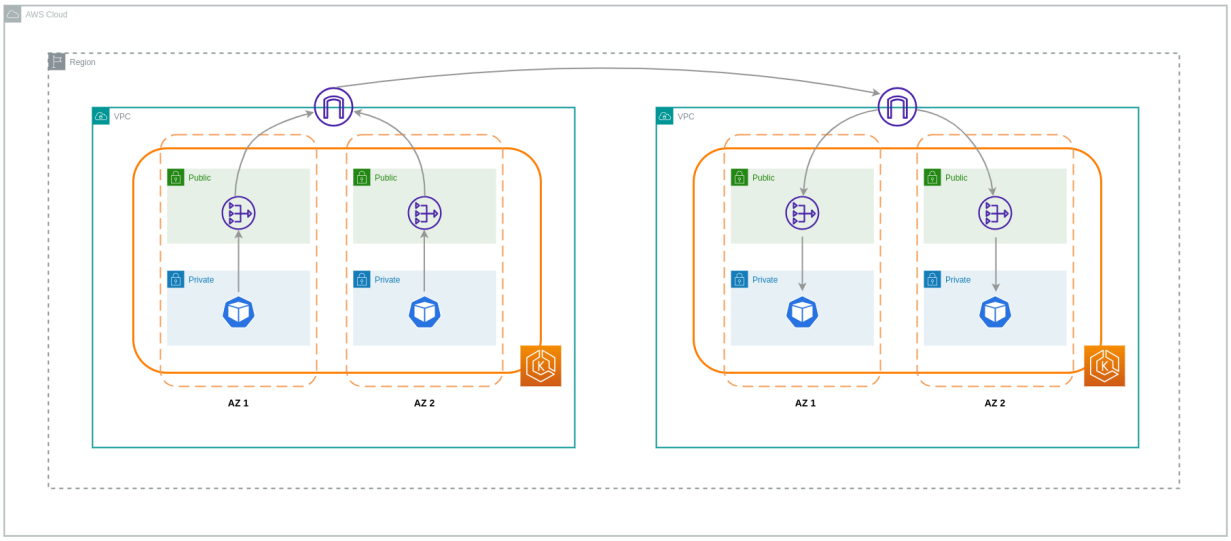
VPC 终端节点按小时收费，通过底层 ENI 进行数据处理需要支付额外费用。请注意，可用区间数据传输不收费。

下图显示了通过 VPC 终端节点与 AWS 服务通信的 Pod。




## 之间的数据传输 VPCs

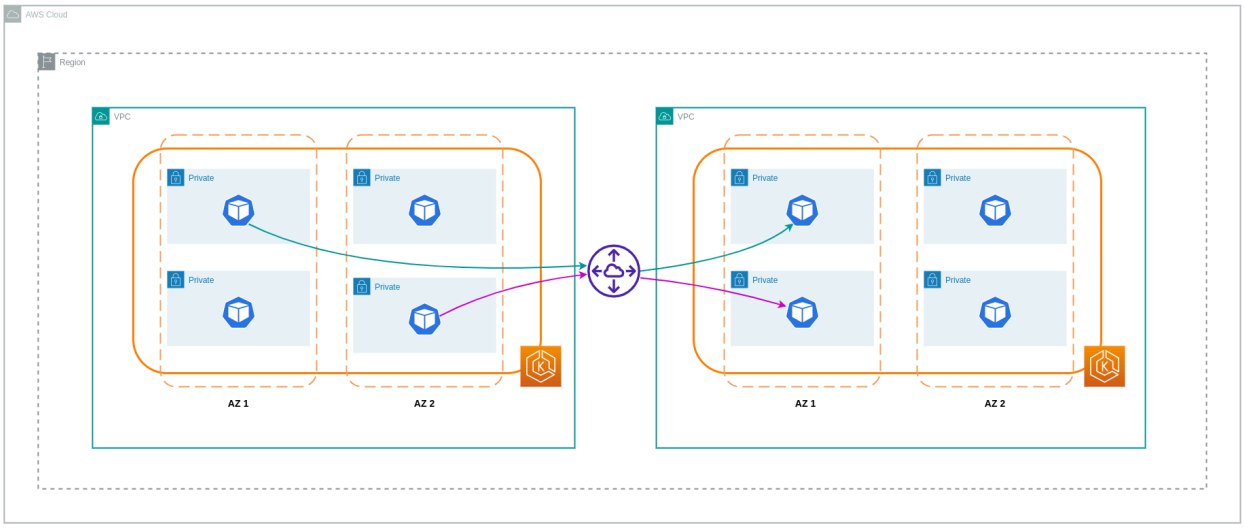
在某些情况下，不同的 VPC（在同一 AWS 区域内）中可能有工作负载需要相互通信。这可以通过允许流量通过连接到相应VPC的互联网网关穿越公共互联网来实现。可以通过在公有子网中部署 EC2 实例、NAT 网关或 NAT 实例等基础设施组件来启用此类通信。但是，包含这些组件的设置将产生 processing/transferring 数据进出 VPCs费用。如果进出分开的流量 VPCs 正在流过 AZs，则数据传输将额外收费。下图描绘了一种使用 NAT 网关和 Internet 网关在不同 VPCs工作负载之间建立通信的设置。




## VPC 对等连接

要降低此类用例的成本，您可以使用 VPC 对等连接。使用 VPC 对等连接时，停留在同一可用区内的网络流量无需支付数据传输费用。如果交通过境 AZs，则会产生费用。不过，建议使用 VPC 对等互连方法，以便在同一 AWS 区域 VPCs 内的不同工作负载之间进行经济高效的通信。但是，需要注意的是，VPC 对等互连主要对 1:1 VPC 连接有效，因为它不允许传输网络。

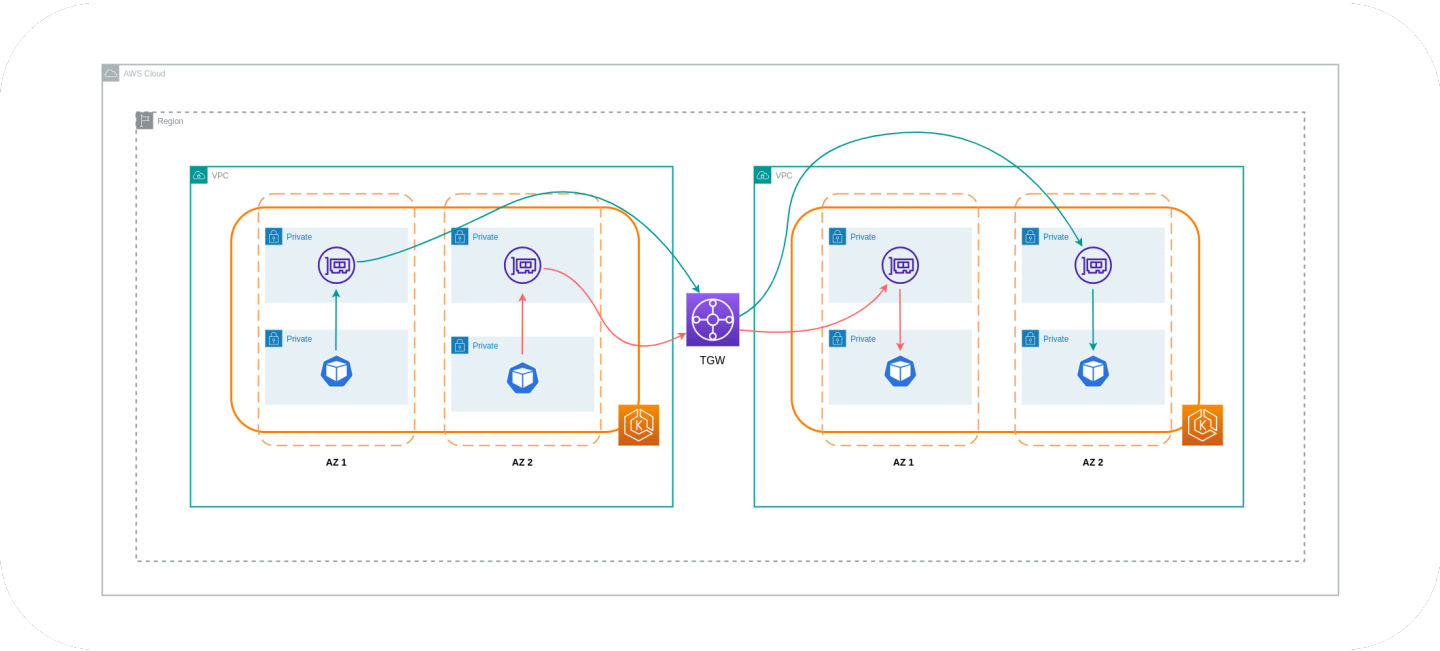
下图是通过 VPC 对等连接进行工作负载通信的高级表示。



### 可传递网络连接

如上一节所述，VPC 对等连接不允许传输网络连接。如果您想根据过渡网络要求连接 3 个或更多个 VPCs ，则应使用 T ransit Gateway (TGW)。这将使您能够克服 VPC 对等互连的限制或与在多个 VPC 对等连接之间建立多个 VPC 对等连接相关的任何运营开销。 VPCs您按小时计费，并按发送到 TGW 的数据收费。流经TGW的可用区间流量不产生任何目标成本。

下图显示了不同 VPCs 但位于相同 AWS 区域内的不同工作负载之间通过 TGW 的可用区间流量。




## 使用服务网格

服务网格提供强大的联网功能，可用于降低 EKS 集群环境中的网络相关成本。但是，如果您采用服务网格，则应仔细考虑服务网格会给您的环境带来的操作任务和复杂性。

### 限制流向可用区的流量

#### 使用 Istio 的局部加权分布

Istio 使您能够在路由发生后将网络策略应用于流量。这是使用目的地规则（例如地点加权分布）完成的。使用此功能，您可以根据流量来源控制可以前往某个目的地的流量的权重（以百分比表示）。这种流量的来源可以来自外部（或面向公众）的负载均衡器，也可以来自集群本身的 Pod。当所有 Pod 端点都可用时，将根据加权轮询负载平衡算法选择地点。如果某些端点运行状况不佳或不可用，则会自动调整区域权重，以反映可用端点的这种变化。

> **Note**
>
> 在实施区域加权分布之前，您应该首先了解您的网络流量模式以及目标规则策略可能对应用程序行为产生的影响。因此，使用诸如 AWS X-Ray 或 Jaeg er 之类的工具建立分布式跟踪机制非常重要。

上面详述的 Istio 目标规则也可以应用于管理从负载均衡器到 EKS 集群中 Pod 的流量。本地加权分配规则可以应用于从高可用性负载均衡器（特别是 Ingress Gateway）接收流量的服务。这些规则允许您根据其区域来源（在本例中为负载均衡器）来控制流量流向何处。如果配置正确，与将流量均匀或随机分配到不同的 Pod 副本的负载均衡器相比，跨区域的出口流量将更少。 AZs

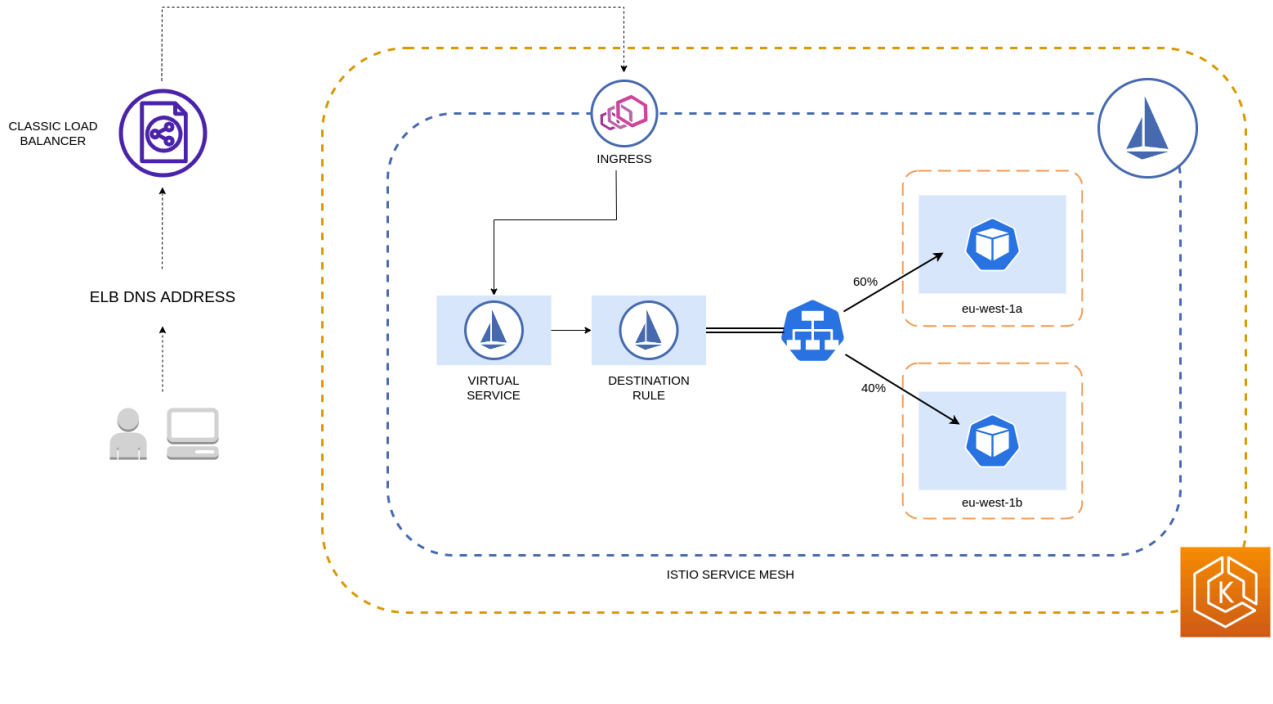
以下是 Istio 中目标规则资源的代码块示例。如下所示，该资源为来自该区域 3 个不同 AZs eu-west-1地区的传入流量指定了加权配置。这些配置声明，来自给定可用区的大部分传入流量（在本例中为 70%）应代理到其来源的同一个可用区中的目的地。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: express-test-dr
spec:
  host: express-test.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:                      +
      localityLbSetting:
        distribute:
        - from: eu-west-1/eu-west-1a/  +
          to:
            "eu-west-1/eu-west-1a/_": 70
            "eu-west-1/eu-west-1b/_": 20
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1b/_  +
          to:
            "eu-west-1/eu-west-1a/_": 20
            "eu-west-1/eu-west-1b/_": 70
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1c/_  +
          to:
            "eu-west-1/eu-west-1a/_": 20
            "eu-west-1/eu-west-1b/_": 10
            "eu-west-1/eu-west-1c/*": 70**
```

```
    connectionPool:
      http:
        http2MaxRequests: 10
        maxRequestsPerConnection: 10
    outlierDetection:
      consecutiveGatewayErrors: 1
      interval: 1m
      baseEjectionTime: 30s
```

> **Note**
>
> 可以分配到目的地的最小权重为 1%。这样做的原因是为了维护故障转移区域和区域，以防主目标中的终端节点运行状况不佳或不可用。

下图描绘了一个场景，其中 eu -west-1 区域中有一个高可用性负载均衡器，并应用了地区加权分布。此图表的目标规则策略配置为将来自 eu-west-1a 的流量的 60% 发送到同一个可用区中的 Pod，而来自 e u-west-1a 的流量中有 40% 应发送到 e u-west- 1b 中的 Pod。



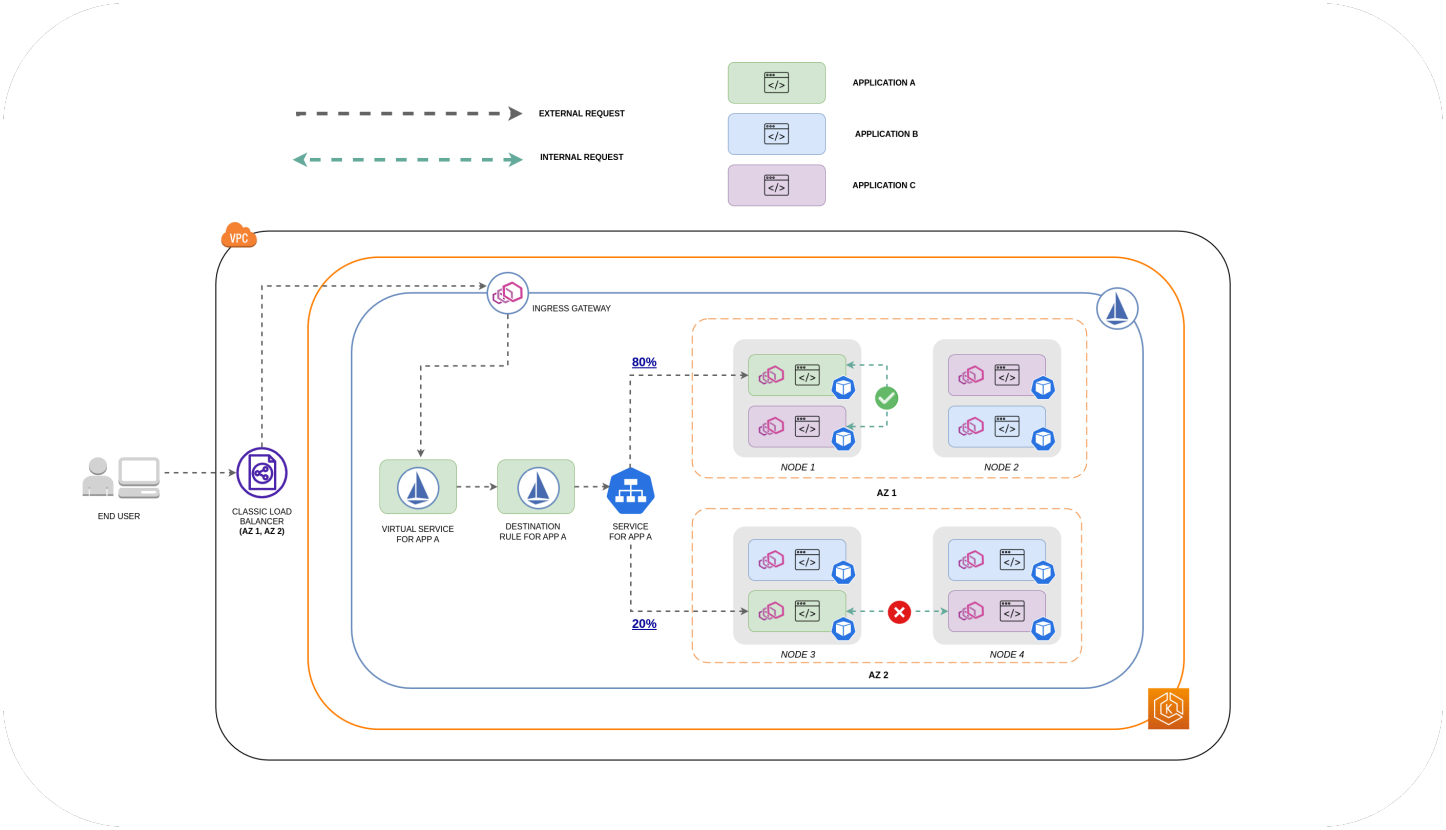

## 限制流向可用区和节点的流量

### 在 Istio 中使用服务内部流量策略

为了降低与外部传入流量和 Pod 之间的内部流量相关的网络成本，您可以将 Istio 的目标规则和 Kubernetes 服务内部流量策略结合起来。将 Istio 目标规则与服务内部流量策略相结合的方法在很大程度上取决于三件事：

- 微服务的作用
- 跨微服务的网络流量模式
- 如何在 Kubernetes 集群拓扑中部署微服务

下图显示了嵌套请求情况下的网络流会是什么样子，以及上述策略将如何控制流量。



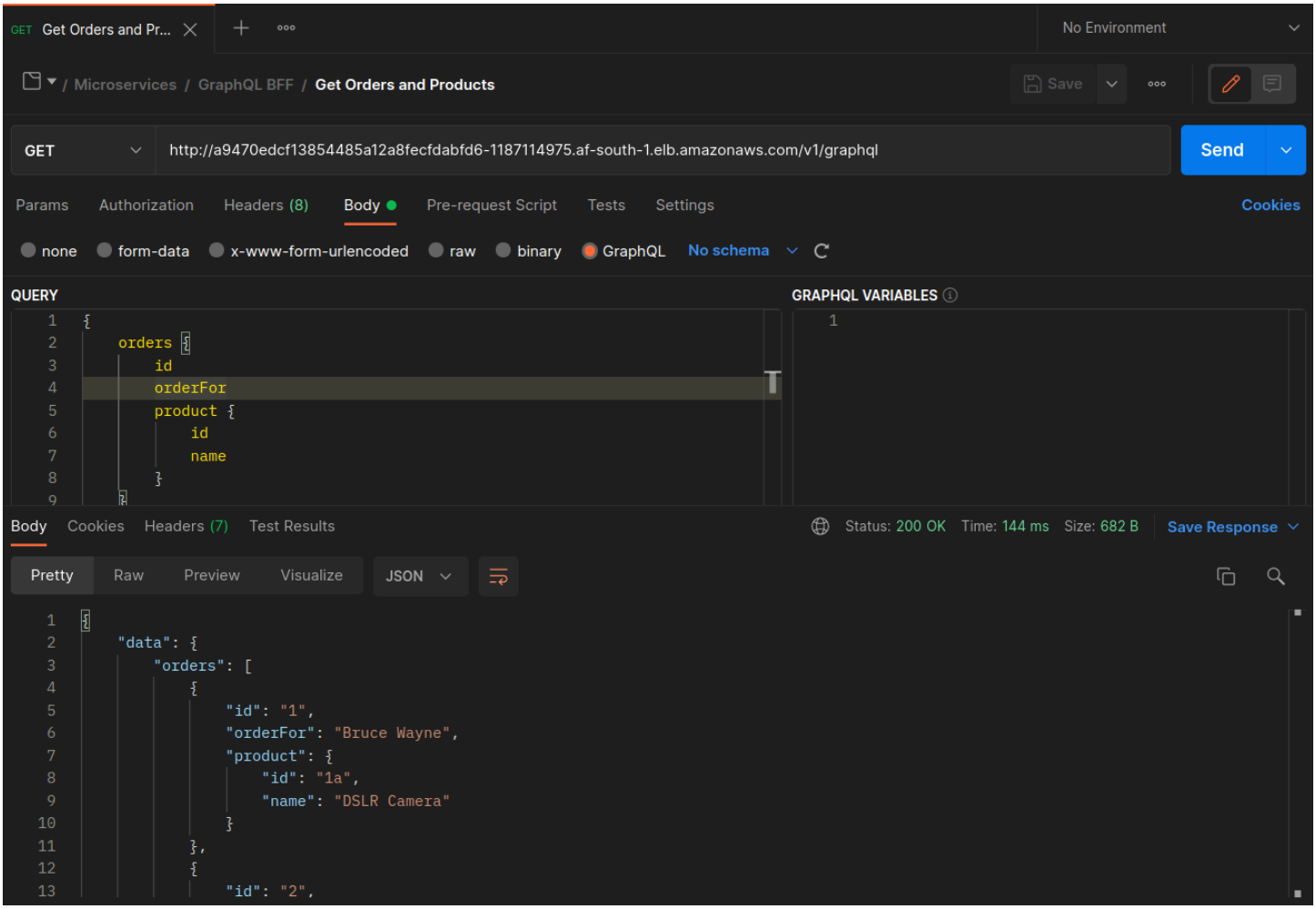

1. 最终用户向应用程序 A 发出请求，而应用程序 A 又向 AP C 发出嵌套请求。 此请求首先发送到高可用性负载均衡器，该负载均衡器在可用区 1 和可用区 2 中都有实例，如上图所示。
2. 然后，Istio 虚拟服务会将外部传入的请求路由到正确的目的地。
3. 请求被路由后，Istio 目标规则 AZs 根据请求的来源（AZ 1 或 AZ 2）控制流向相应请求的流量。
4. 然后，流量会进入应用程序 A 的服务，然后被代理到相应的 Pod 端点。如图所示，80% 的传入流量发送到可用区 1 中的 Pod 终端节点，20% 的传入流量发送到可用区 2。
5. 然后，应用程序 A 向应用程序 C 发出内部请求。 APP C 的服务启用了内部流量策略 (`internalTrafficPolicy`: Local`)。

6. 由于 APP C 有可用的节点本地端点，因此从 APP A（在节点 1 上）向 AP P C 发出的内部请求是成功的。
7. 从应用程序 A（在节点 3 上）向应用程序 C 发出的内部请求失败，因为应用程序 C 没有可用的节点本地端点。 如图所示，应用程序 C 在 NODE 3 上没有副本。 *

以下屏幕截图是从这种方法的真实示例中截取的。第一组屏幕截图演示了向节点ip-10-0-0-151.af-south-1.compute.internal上托管orders副本的成功外部请求graphql以及来自的成功嵌套请求。graphql

test-results.txt

```
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-0-147.af-south-1.compute.i
NAME                        READY   STATUS    RESTARTS   AGE   IP          NODE
products-8567b458c8-brgdp   2/2     Running   0          64m   10.0.0.83   ip-10-0-0-147.af-south-1.com
------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-1-153.af-south-1.compute.i
NAME                        READY   STATUS    RESTARTS   AGE   IP          NODE
products-8567b458c8-5shq8   2/2     Running   0          65m   10.0.1.4    ip-10-0-1-153.af-south-1.comp
------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-1-68.af-south-1.compute.in
NAME                      READY   STATUS    RESTARTS   AGE   IP           NODE
orders-684bfbd6d9-5dxcf   2/2     Running   0          22m   10.0.1.145   ip-10-0-1-68.af-south-1.compu
------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-0-151.af-south-1.compute.i
NAME                      READY   STATUS    RESTARTS   AGE   IP           NODE
graphql-8f877c686-k7k9b   2/2     Running   0          18m   10.0.0.239   ip-10-0-0-151.af-south-1.comp
orders-684bfbd6d9-rc2zp   2/2     Running   0          18m   10.0.0.182   ip-10-0-0-151.af-south-1.comp
```




使用 Istio，您可以验证和导出代理知道的任何上游集群和终端节点的统计信息。这可以帮助了解网络流以及工作负载各服务之间的分布份额。继续使用相同的示例，可以使用以下命令获取graphql代理知道的orders端点：

```
kubectl exec -it deploy/graphql -n ecommerce -c istio-proxy -- curl localhost:15000/
clusters | grep orders
```

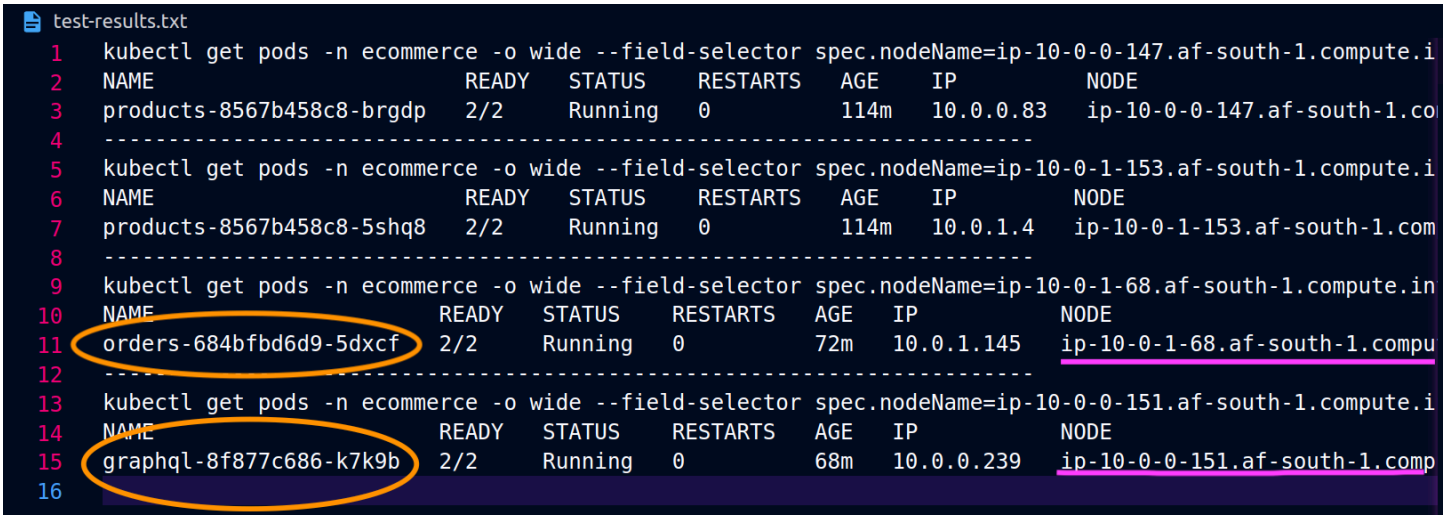
```
...
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_error::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_success::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_timeout::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_total::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**health_flags::healthy**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**region::af-south-1**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**zone::af-south-1b**
...
```

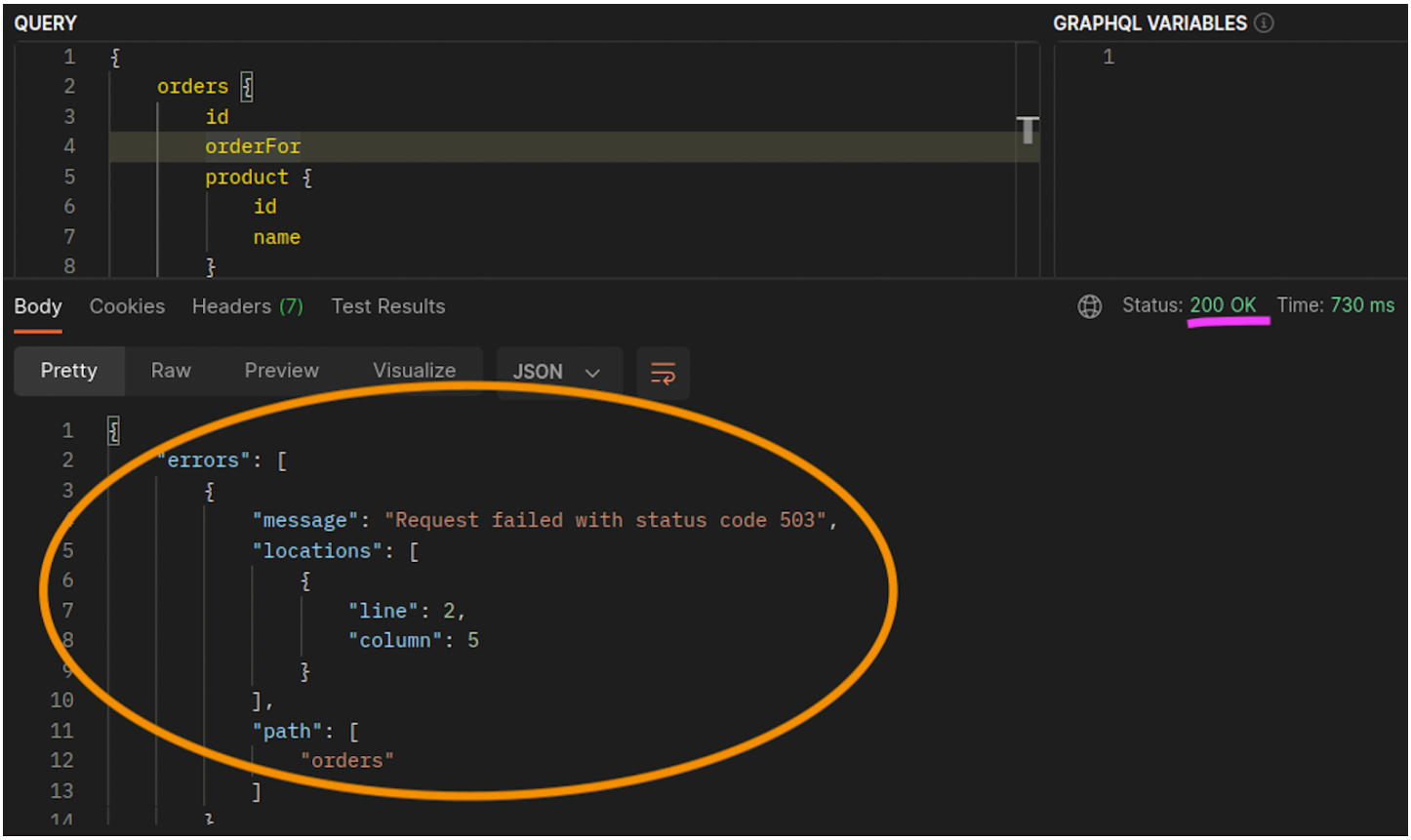
在这种情况下，graphql代理只知道与其共享节点的副本的orders终端节点。如果您从 orders Service 中删除该internalTrafficPolicy: Local设置，然后重新运行类似于上面的命令，则结果将返回分布在不同节点上的副本的所有端点。此外，通过检查相应的rq_total端点，您会注意到网络分布的份额相对均匀。因此，如果端点与在不同区域中运行的上游服务相关联 AZs，则这种跨区域的网络分布将导致更高的成本。

如上文前一节所述，你可以利用 pod 的亲和力将频繁通信的 Pod 放在同一个地方。

```
...
spec:
...
  template:
    metadata:
      labels:
        app: graphql
        role: api
        workload: ecommerce
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
      nodeSelector:
        managedBy: karpenter
        billing-team: ecommerce
...
```

当graphql和orders副本不在同一个节点 (ip-10-0-0-151.af-south-1.compute.internal) 上共存时，第一个请求成功，如下面graphql的 Postman 屏幕截图所200 response code示，而第二个从graphql到的嵌套请求orders失败，结果为 a。503 response code




## 其他资源

- 使用 Istio 解决 EKS 上的延迟和数据传输成本
- 探索拓扑感知提示对亚马逊 Elastic Kubernetes Service 中网络流量的影响 Amazon Kubernetes Service
- 查看你的 Amazon EKS 跨可用区容器到容器网络字节

- 使用 Istio 优化可用区流量
- 使用拓扑感知路由优化 AZ 流量
- 利用服务内部流量策略优化 Kubernetes 的成本和性能
- 利用 Istio 和服务内部流量策略优化 Kubernetes 的成本和性能
- Overview of Data Transfer Costs for Common Architectures
- 了解 AWS 容器服务的数据传输成本

# 存储

## 概览

在某些情况下，您可能需要运行需要在短期或长期基础上保留数据的应用程序。对于此类用例，可以由 Pod 定义和挂载卷，这样它们的容器就可以利用不同的存储机制。Kubernetes 支持不同类型的卷用于临时存储和永久存储。存储的选择在很大程度上取决于应用程序要求。每种方法都有成本影响，下面详细介绍的做法将帮助您在 EKS 环境中为需要某种形式存储的工作负载实现成本效益。

## 短暂卷

临时卷适用于需要临时本地卷但不需要在重启后保留数据的应用程序。这方面的例子包括对暂存空间、缓存和只读输入数据（如配置数据和机密）的要求。你可以在这里找到 Kubernetes 临时卷的更多细节。大多数临时卷（例如 emptyDir、ConfigMap、downwardAPI、secret、hostpath）都由本地连接的可写入设备（通常是根磁盘）或 RAM 提供支持，因此选择最具成本效益和性能的主机卷非常重要。

### 使用 EBS 卷

我们建议从 gp3 作为主机根卷开始。它是Amazon EBS提供的最新通用固态硬盘容量，与gp2卷相比，每GB的价格也更低（最高可达20%）。

### 使用亚马逊 EC2 实例存储

Amazon EC2 实例存储为您的 EC2 实例提供临时块级存储。 EC2 实例存储提供的存储可通过物理连接到主机的磁盘进行访问。与 Amazon EBS 不同，您只能在实例启动时附加实例存储卷，而且这些卷仅在实例的生命周期内存在。它们不能分离并重新连接到其他实例。您可以在此处了解有关 Amazon EC2 实例存储的更多信息。实例存储容量不收取任何额外费用。这使得它们（实例存储卷）比具有大 EBS 卷的普通 EC2 实例更具成本效益。

要在 Kubernetes 中使用本地存储卷，您应该使用 Amazon EC2 用户数据对磁盘进行分区、配置和格式化，以便可以按照 pod 规范安装卷。HostPath或者，您可以利用本地永久卷静态配置器来简化本地

存储管理。本地永久卷静态配置器允许您通过标准 Kubernetes PersistentVolumeClaim (PVC) 接口访问本地实例存储卷。此外，它还会配置 PersistentVolumes (PVs)，其中包含节点亲和性信息，将 Pod 调度到正确的节点。尽管它使用 Kubernetes PersistentVolumes，但 EC2 实例存储卷本质上是短暂的。写入临时磁盘的数据仅在实例的生命周期内可用。当实例终止时，数据也会终止。有关更多详细信息，请参阅此博客。

请记住，在使用 Amazon EC2 实例存储卷时，总的 IOPS 限制与主机共享，并将 Pod 绑定到特定的主机。在采用 Amazon EC2 实例存储卷之前，您应该仔细检查您的工作负载要求。

## 永久卷

Kubernetes 通常与运行无状态应用程序相关联。但是，在某些情况下，您可能需要运行微服务，这些微服务需要将一个请求中的永久数据或信息保留到下一个请求。数据库是此类用例的常见示例。但是，Pod 及其中的容器或进程本质上是短暂的。要在 Pod 的生命周期之后保留数据，您可以使用定义 PVs 对独立于 Pod 的特定位置的存储的访问权限。与 PVs 之相关的成本在很大程度上取决于所使用的存储类型以及应用程序的使用方式。

此处列出了支持 PVs Amazon EKS 上的 Kubernetes 的不同类型的存储选项。下面介绍的存储选项是亚马逊 EBS、Amazon EFS、Amazon for Lustre、Ama FSx zon for ONTA FSx P NetApp 。

### 亚马逊 Elastic Block Store (EBS) Volumes

Amazon EBS 卷可以作为 Kubernetes 使用， PVs 以提供块级存储卷。它们非常适合依赖随机读取和写入的数据库，以及执行长时间、连续读取和写入的吞吐量密集型应用程序。Amazon Elastic Block Store 容器存储接口 (CSI) 驱动程序允许 Amazon EKS 集群管理永久卷的 Amazon EBS 卷的生命周期。容器存储接口支持并促进了 Kubernetes 与存储系统之间的交互。将 CSI 驱动程序部署到您的 EKS 集群后，您可以通过原生 Kubernetes 存储资源访问其功能，例如永久卷 (PVs)、永久卷声明 (PVCs) 和存储类别 ()。SCs此链接提供了如何使用亚马逊 EBS CSI 驱动程序与亚马逊 EBS 卷进行交互的实际示例。

#### 选择合适的音量

我们建议使用最新一代的块存储 (gp3)，因为它在价格和性能之间提供了适当的平衡。它还允许您独立于卷大小扩展卷 IOPS 和吞吐量，而无需配置额外的块存储容量。如果您目前正在使用 gp2 卷，我们强烈建议您迁移到 gp3 卷。博客文章将 Amazon EKS 集群从 gp2 迁移到 gp3 EBS 卷解释了如何使用 CSI 卷快照功能从亚马逊 EKS 集群上的 gp3 上的 gp 2 迁移，并使用需要应用程序停机的 CSI 卷快照功能进行备份和恢复。

Amazon EBS 允许在线更改卷特征，例如卷大小、IOPS 和吞吐量。利用此功能，可以使用 P VC 注释（如本博客中所述）（需要 EBS CSI 驱动程序 v1.19.0+），或者使用此处描述的 API 从 Amazon EKS v1.31 和 EBS CSI 驱动程序 1. 3 5 开始迁移，而无需应用程序停机。VolumeAttributesClass

当您的应用程序需要更高的性能并且需要的卷大于单个 gp3 卷所能支持的容量时，您应该考虑使用 io2 block express。这种类型的存储非常适合规模最大、最 I/O 密集和任务关键型部署，例如 SAP HANA 或其他具有低延迟要求的大型数据库。请记住，实例的 EBS 性能受实例性能限制的限制，因此并非所有实例都支持 io2 block express 卷。您可以在本文档中查看支持的实例类型和其他注意事项。

一个 gp3 卷最多可以支持最大 16,000 个 IOPS、 MiB/s 最大吞吐量 1,000、最大 16TiB。最新一代的预配置 IOPS 固态硬盘卷，可提供高达 256,000 IOPS、4,000 MiB/s、吞吐量和 64TiB。

在这些选项中，您应该根据应用程序的需求量身定制存储性能和成本。

随着时间的推移进行监控和优化

重要的是要了解应用程序的基准性能并监控所选卷的基准性能，以检查其是否满足您的要求 requirements/expectations 或是否过度配置（例如，未充分利用预配置 IOPS 的情况）。

与其从一开始就分配大卷，不如在积累数据时逐渐增加卷的大小。您可以使用 Amazon Elastic Block Store CSI 驱动程序 () aws-ebs-csi-driver 中的卷大小调整功能动态调整卷大小。请记住，您只能增加 EBS 卷的大小。

要识别和移除任何悬挂的 EBS 卷，您可以使用 AW S trusted Advisor 的成本优化类别。此功能可帮助您识别未连接的卷或一段时间内写入活动非常低的卷。有一个名为 Popeye 的云原生开源只读工具，它可以扫描实时 Kubernetes 集群，并报告已部署资源和配置的潜在问题。例如，它可以扫描未使用的 PVs 和 PVCs 并检查它们是否已绑定或是否存在任何卷装入错误。

要深入了解监控，请参阅 EKS 成本优化可观察性指南。

你可以考虑的另一个选择是 AWS Compute Optimizer Amazon EBS 交易量建议。该工具可自动识别所需的最佳卷配置和正确的性能级别。例如，根据过去 14 天的最大利用率，它可以用于与预配置 IOPS、卷大小和 EBS 卷类型有关的最佳设置。它还量化了根据其建议可能节省的每月成本。您可以查看此博客以获取更多详细信息。

备份保留政策

您可以通过拍摄 point-in-time快照来备份 Amazon EBS 卷上的数据。Amazon EBS CSI 驱动程序支持卷快照。您可以使用此处概述的步骤学习如何创建快照和恢复 EBS PV。

后续快照是增量备份，这意味着仅保存设备上在最近一次快照后发生更改的块。由于无需复制数据，这将最大限度缩短创建快照所需的时间和增加存储成本节省。但是，在没有适当的保留策略的情况下

增加旧 EBS 快照的数量可能会在大规模运营时造成意想不到的成本。如果您通过 AWS API 直接备份亚马逊 EBS 卷，则可以利用亚马逊数据生命周期管理器 (DLM)，它为亚马逊弹性区块存储 (EBS) Elastic Block Store 快照和 EBS 支持的亚马逊系统映像 () 提供基于策略的自动生命周期管理解决方案 ()。AMIs通过控制台，可以更轻松地自动创建、保留和删除 EBS 快照和 AMIs。

> **Note**
>
> 目前无法通过亚马逊 EBS CSI 驱动程序使用亚马逊 DLM。

在 Kubernetes 环境中，你可以利用名为 Velero 的开源工具来备份 EBS 永久卷。在安排任务以使备份过期时，您可以设置 TTL 标志。以下是Velero的指南作为示例。

## Amazon Elastic File System (EFS)

Amazon Elastic File System (EFS) 是一个无服务器、完全弹性的文件系统，允许您使用标准文件系统接口和文件系统语义为各种工作负载和应用程序共享文件数据。工作负载和应用程序的示例包括 Wordpress 和 Drupal、JIRA 和 Git 等开发者工具，以及 Jupyter 等共享笔记本系统以及主目录。

Amazon EFS 的主要优势之一是，它可以由分布在多个节点和多个可用区的多个容器进行安装。另一个好处是，您只需为使用的存储空间付费。EFS 文件系统将在您添加和删除文件时自动增长和缩小，从而无需进行容量规划。

要在 Kubernetes 中使用 Amazon EFS，你需要使用亚马逊弹性文件系统 Elastic File System 容器存储接口 (CSI) 驱动程序。aws-efs-csi-driver目前，驱动程序可以动态创建接入点。但是，必须先配置 Amazon EFS 文件系统，并将其作为输入提供给 Kubernetes 存储类参数。

选择正确的 EFS 存储类别

Amazon EFS 提供四种存储类别。

两个标准存储类别：

- 亚马逊 EFS 标准
- 亚马逊 EFS 标准——不频繁访问（EFS 标准-IA）

两个单区域存储类别：

- 亚马逊 EFS One Zone
- 亚马逊 EFS 单区-不频繁访问 (EFS 单区-IA)

不频繁访问 (IA) 存储类针对并非每天都访问的文件进行了成本优化。借助 Amazon EFS 生命周期管理，您可以将生命周期策略期间（7、14、30、60 或 90 天）未被访问的文件移动到 IA 存储类别，与 EFS 标准和 EFS One Zone 存储类别相比，存储成本可分别降低多达 92%。

借助 EFS Intelligent-Tiering，生命周期管理可以监控文件系统的访问模式，并自动将文件移动到最佳存储类别。

> **Note**
>
> aws-efs-csi-driver 目前无法控制更改存储类别、生命周期管理或智能分层。应在 AWS 控制台中手动设置或通过 EFS 进行设置 APIs。

> **Note**
>
> aws-efs-csi-driver 与基于 Windows 的容器镜像不兼容。

> **Note**
>
> 启用 vol-metrics-opt-in（发出音量指标）时存在已知的内存问题，这是因为该DiskUsage功能消耗的内存量与文件系统的大小成正比。目前，我们建议在大型文件系统上禁用 vol-metrics-opt-in `--`选项，以免消耗太多内存。以下是 github 问题链接，了解更多详情。

## 亚马逊 f FSx or Lustre

Lustre 是一种高性能并行文件系统，通常用于需要高达数百的吞吐量 GB/s 和亚毫秒的每次操作延迟的工作负载。它用于机器学习训练、财务建模、HPC 和视频处理等场景。Amazon FSx for Lustre 提供完全托管的共享存储，具有可扩展性和性能，并与 Amazon S3 无缝集成。

你可以使用 Amazon EKS 的 for Lustre CSI 驱动程序或 AWS 上 FSx 自行管理的 Kubernetes 集群，使用由 for Lustre FSx 提供的 Kubernetes 永久存储卷。有关更多详细信息和示例，请参阅 Amazon EKS 文档。

链接到亚马逊 S3

建议将驻留在 Amazon S3 上的高度持久的长期数据存储库与 for Lustre 文件系统关联起来。 FSx 链接后，大型数据集将根据需要从 Amazon S3 延迟加载到 FSx Lustre 文件系统。您也可以将分析和结果运行回 S3，然后删除 Lustre 文件系统。

选择正确的部署和存储选项

FSx for Lustre 提供了不同的部署选项。第一个选项叫做 s cratch，它不复制数据，而第二个选项叫做 persi st ent，顾名思义，它可以保留数据。

第一个选项（从头开始）可用于降低临时短期数据处理的成本。永久部署选项专为长期存储而设计，可自动复制 AWS 可用区域内的数据。它还支持固态硬盘和硬盘存储。

你可以在 lustre 文件系统的 Kubernetes FSx 的参数下配置所需的部署类型。 StorageClass以下是提供示例模板的链接。

> **Note**
>
> 对于延迟敏感型工作负载或需要最高 IOPS/吞吐量级别的工作负载，您应该选择 SSD 存储。对于以吞吐量为重点且对延迟不敏感的工作负载，您应该选择 HDD 存储。

启用数据压缩

也可以通过将 “LZ4” 指定为数据压缩类型，在文件系统上启用数据压缩。启用后，所有新写入的文件将在写入磁盘之前自动压缩 Lustre，并在读取时解压缩。 FSx LZ4 数据压缩算法是无损的，因此可以从压缩的数据中完全重建原始数据。

你可以在 for lustre 文件系统的 Kubernet FSx es 的参数 LZ4 下配置数据压缩类型。 StorageClass当该值设置为 NONE（默认值）时，压缩功能处于禁用状态。此链接提供示例模板。

> **Note**
>
> Amazon FSx for Lustre 与基于 Windows 的容器镜像不兼容。

## FSx 适用于 NetApp ONTAP 的亚马逊

Amazon FSx f NetApp or ONTAP 是一个完全托管的共享存储，建立在 ONTAP 文件系统 NetApp 之上。 FSx for ONTAP 提供功能丰富、快速、灵活的共享文件存储，可通过在 AWS 或本地运行的 Linux、Windows 和 macOS 计算实例进行广泛访问。

Amazon FSx NetApp for ONTAP 支持两个存储层：1/主层和 2/容量池层。

主层是基于固态硬盘的高性能预配置层，用于存放对延迟敏感的活跃数据。完全弹性的容量池层针对不经常访问的数据进行了成本优化，可随着数据分层到该层而自动扩展，并提供几乎无限的 PB 容量。您

可以在容量池存储上启用数据压缩和重复数据删除，从而进一步减少数据消耗的存储容量。 NetApp基于策略的本机 FabricPool 功能可持续监控数据访问模式，自动在存储层之间双向传输数据以优化性能和成本。

NetApp的 Astra Trident 使用 CSI 驱动程序提供动态存储编排，该驱动程序允许 Amazon EKS 集群管理亚马逊支持的 FSx 适用于 NetApp ONTAP 文件 PVs 系统的永久卷的生命周期。要开始使用，请参阅 Astra Trident 文档中的在亚马逊 FSx 上使用 Astra Trident for NetApp ONTAP。

## 其他考虑因素

### 最小化容器镜像的大小

部署容器后，容器镜像将作为多层缓存在主机上。通过减小图像的大小，可以减少主机上所需的存储量。

通过从一开始就使用精简版的基础映像，例如暂存映像或无发行版容器镜像（仅包含您的应用程序及其运行时依赖项），除了可以减少攻击面积和缩短图像提取时间等其他附带优势外，还可以降低存储成本。

您还应该考虑使用开源工具，例如 Slim.ai，它提供了一种简单、安全的方法来创建最少的图像。

多层包、工具、应用程序依赖关系、库可以很容易地膨胀容器镜像的大小。通过使用多阶段构建，您可以有选择地将工件从一个阶段复制到另一个阶段，将最终图像中不需要的所有内容排除在外。您可以在此处查看更多图像构建最佳实践。

需要考虑的另一件事是将缓存的图像保留多长时间。当使用一定数量的磁盘时，您可能需要从图像缓存中清理陈旧的图像。这样做将有助于确保有足够的空间供主机操作。默认情况下，kubelet 每五分钟对未使用的图像执行一次垃圾回收，每分钟对未使用的容器执行一次垃圾回收。

要为未使用的容器和镜像垃圾收集配置选项，请使用配置文件调整 kubelet，并使用*KubeletConfiguration*资源类型更改与垃圾收集相关的参数。

你可以在 Kubernetes 文档中了解更多相关信息。

# 可观察性

## 简介

可观测性工具可帮助您高效检测、修复和调查您的工作负载。随着您使用EKS的增加，遥测数据的成本自然会增加。有时，要平衡运营需求和衡量对业务至关重要的因素并控制可观察性成本，可能是一项艰

巨的任务。本指南重点介绍可观测性的三个支柱的成本优化策略：日志、指标和跟踪。这些最佳实践中的每一种都可以独立应用，以适应组织的优化目标。

## 日志记录

日志在监控集群中的应用程序并对其进行故障排除方面起着至关重要的作用。有几种策略可以用来优化日志成本。下面列出的最佳实践策略包括检查您的日志保留策略以对日志数据的保留时间实施精细控制，根据重要性将日志数据发送到不同的存储选项，以及利用日志筛选来缩小存储的日志消息的类型。高效管理日志遥测可以为您的环境节省成本。

## EKS 控制飞机

### 优化您的控制平面日志

Kubernetes 控制平面是一组用于管理集群的组件，这些组件将不同类型的信息作为日志流发送到 Amazon 中的日志组。 CloudWatch虽然启用所有控制平面日志类型有好处，但您应该了解每个日志中的信息以及存储所有日志遥测数据的相关成本。对于从您的集群发送到 Ama CloudWatch zon CloudWatch Logs 的日志，您需要支付标准日志数据摄取和存储费用。在启用它们之前，请评估是否需要每个日志流。

例如，在非生产集群中，有选择地启用特定的日志类型，例如 api 服务器日志，仅用于分析，然后停用。但是对于生产集群，您可能无法重现事件，并且解决问题需要更多的日志信息，则可以启用所有日志类型。更多控制平面成本优化的实施细节请参见这篇博客文章。

将日志流式传输到 S3

另一种成本优化最佳实践是通过 CloudWatch 日志订阅将控制平面日志流式传输到 S3。利用 CloudWatch 日志订阅，您可以有选择地将日志转发到 S3，与无限期保留日志相比，S3 提供了更具成本效益的长期存储。 CloudWatch例如，对于生产集群，您可以创建一个关键日志组，并利用订阅在 15 天后将这些日志流式传输到 S3。这将确保您可以快速访问日志进行分析，还可以通过将日志移动到更具成本效益的存储空间来节省成本。

> **Important**
>
> 自 2023 年 9 月 5 日起，EKS 日志在 Amazon 日志中被归类为销售日志。 CloudWatch Vended Logs 是 AWS 服务代表客户在本地发布的特定 AWS 服务日志，按批量折扣定价提供。请访问 Amazon CloudWatch 定价页面，了解有关 Vended Logs 定价的更多信息。

# EKS 数据平面

## 日志保留

Amazon CloudWatch 的默认保留政策是无限期保留日志且永不过期，这会产生适用于您的 AWS 地区的存储成本。为了降低存储成本，您可以根据工作负载要求为每个日志组自定义保留策略。

在开发环境中，可能不需要很长的保留期。但是在生产环境中，您可以设置更长的保留期策略以满足故障排除、合规性和容量规划要求。例如，如果您在假日旺季运行电子商务应用程序，则系统负载较重，并且可能会出现可能无法立即注意到的问题，则需要设置更长的日志保留时间，以便进行详细的故障排除和事件后分析。

您可以在 AWS CloudWatch 控制台或 AW S API 中根据每个日志组将保留期配置为 1 天到 10 年不等。灵活的保留期可以节省日志存储成本，同时还可以维护关键日志。

## 日志存储选项

存储是可观测性成本的重要驱动因素，因此优化日志存储策略至关重要。您的策略应与您的工作负载需求保持一致，同时保持性能和可扩展性。降低存储日志成本的一种策略是利用 AWS S3 存储桶及其不同的存储层。

### 将日志直接转发到 S3

考虑将不太重要的日志（例如开发环境）直接转发到 S3 而不是 Cloudwatch。这可能会对日志存储成本产生直接影响。一种选择是使用 Fluentbit 将日志直接转发到 S3。您可以在 “ FluentBit 传输容器日志以进行保留的目的地” 一[OUTPUT]节中对此进行定义。在此处查看其他配置参数。

```
[OUTPUT]
        Name eks_to_s3
        Match application.*
        bucket $S3_BUCKET name
        region us-east-2
        store_dir /var/log/fluentbit
        total_file_size 30M
        upload_timeout 3m
```

### 将日志转发到 CloudWatch 仅用于短期分析

对于更重要的日志，例如可能需要对数据进行即时分析的生产环境，可以考虑将日志转发到 CloudWatch。您可以在 “ FluentBit 传输容器日志以进行保留的目的地” 一[OUTPUT]节中对此进行定义。在此处查看其他配置参数。

```
[OUTPUT]
        Name eks_to_cloudwatch_logs
        Match application.*
        region us-east-2
        log_group_name fluent-bit-cloudwatch
        log_stream_prefix from-fluent-bit-
        auto_create_group On
```

但是，这不会立即影响您的成本节约。为了节省更多费用，您必须将这些日志导出到 Amazon S3。

从以下地址导出到亚马逊 S3 CloudWatch

为了长期存储亚马逊 CloudWatch 日志，我们建议将您的亚马逊 EKS CloudWatch 日志导出到亚马逊简单存储服务 (Amazon S3) Simple S3 Service。您可以通过控制台或 API 创建导出任务，将日志转发到 Amazon S3 存储桶。完成此操作后，Amazon S3 提供了许多选项来进一步降低成本。您可以定义自己的 Amazon S3 生命周期规则，将日志移动到符合您需求的存储类别，或者利用 A mazon S3 智能分层存储类让 AWS 根据您的使用模式自动将数据移至长期存储。有关更多详细信息，请参阅此博客。例如，对于您的生产环境，日志存放时间超过 30 天，然后导出到 Amazon S3 存储桶。 CloudWatch 然后，如果您需要在以后参考日志，则可以使用 Amazon Athena 来查询 Amazon S3 存储桶中的数据。

## 降低日志级别

为您的应用程序练习选择性日志记录。默认情况下，您的应用程序和节点都会输出日志。对于应用程序日志，请调整日志级别，使其与工作负载和环境的重要性保持一致。例如，下面的 java 应用程序正在输出INFO日志，这是典型的默认应用程序配置，根据代码的不同，可能会生成大量的日志数据。

```
import org.apache.log4j.*;

public class LogClass {
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

public static void main(String[] args) {
      log.setLevel(Level.INFO);

   log.debug("This is a DEBUG message, check this out!");
   log.info("This is an INFO message, nothing to see here!");
   log.warn("This is a WARN message, investigate this!");
   log.error("This is an ERROR message, check this out!");
   log.fatal("This is a FATAL message, investigate this!");    } }
```

在开发环境中，将日志级别更改为DEBUG，因为这可以帮助你调试问题，或者在潜在问题进入生产之前抓住它们。

```
      log.setLevel(Level.DEBUG);
```

在生产环境中，可以考虑将日志级别修改为ERROR或FATAL。只有当您的应用程序出现错误时，它才会输出日志，从而减少日志输出，并帮助您专注于有关应用程序状态的重要数据。

```
      log.setLevel(Level.ERROR);
```

你可以微调各种 Kubernetes 组件的日志级别。例如，如果您使用 Bottlerocket 作为 EKS 节点操作系统，则有一些配置设置允许您调整 kubelet 进程日志级别。此配置设置的片段如下所示。请注意，默认日志级别为 2，它会调整进程的日志详细程度。kubelet

```
[settings.kubernetes]
log-level = "2"
image-gc-high-threshold-percent = "85"
image-gc-low-threshold-percent = "80"
```

对于开发环境，您可以将日志级别设置为大于 2 以查看其他事件，这有利于调试。对于生产环境，您可以将级别设置为 0，以便仅查看关键事件。

## 利用过滤器

使用默认 EKS Fluentbit 配置将容器日志发送到 Cloudwatch 时，请 FluentBit 捕获所有富含 Kubernetes 元数据的应用程序容器日志并将其发送到 Cloudwatch，如下面的配置块所示。[INPUT]

```
 [INPUT]
     Name                tail
     Tag                 application.*
     Exclude_Path        /var/log/containers/cloudwatch-agent*, /var/log/containers/
fluent-bit*, /var/log/containers/aws-node*, /var/log/containers/kube-proxy*
     Path                /var/log/containers/*.log
     Docker_Mode         On
     Docker_Mode_Flush   5
     Docker_Mode_Parser  container_firstline
     Parser              docker
     DB                  /var/fluent-bit/state/flb_container.db
     Mem_Buf_Limit       50MB
```

```
    Skip_Long_Lines     On
    Refresh_Interval    10
    Rotate_Wait         30
    storage.type        filesystem
    Read_from_Head      ${READ_FROM_HEAD}
```

上面的[INPUT]部分正在摄取所有容器日志。这可能会生成大量可能不是必需的数据。筛选出这些数据可以减少发送到的日志数据量， CloudWatch 从而降低成本。您可以先对日志应用过滤器，然后再将其输出到 CloudWatch。Fluentbit 在本节中对此进行了定义。[FILTER]例如，过滤掉 Kubernetes 元数据以使其不被附加到日志事件可以减少日志量。

```
[FILTER]
    Name                nest
    Match               application.*
    Operation           lift
    Nested_under        kubernetes
    Add_prefix          Kube.

[FILTER]
    Name                modify
    Match               application.*
    Remove              Kube.<Metadata_1>
    Remove              Kube.<Metadata_2>
    Remove              Kube.<Metadata_3>

[FILTER]
    Name                nest
    Match               application.*
    Operation           nest
    Wildcard            Kube.*
    Nested_under        kubernetes
    Remove_prefix       Kube.
```

## Metrics

指标提供了有关系统性能的宝贵信息。通过将所有与系统相关的或可用的资源指标整合到一个集中的位置，您可以比较和分析性能数据。这种集中式方法使您能够做出更明智的战略决策，例如扩大或缩减资源。此外，指标在评估资源运行状况方面起着至关重要的作用，使您能够在必要时采取主动措施。通常，可观测性成本会随着遥测数据的收集和保留而增加。以下是您可以实施的几种策略来降低公制遥测的成本：仅收集重要的指标，降低遥测数据的基数，以及微调遥测数据收集的粒度。

## 监控重要内容，只收集你需要的东西

第一种降低成本的策略是减少您收集的指标数量，进而降低留存成本。

1. 首先，从您和/或利益相关者的要求向后推进，以确定最重要的指标。每个人的成功指标都不一样！知道好看是什么样子，并据此进行衡量。
2. 考虑深入研究你所支持的工作负载，并确定其关键性能指标 (KPIs)，又名 “黄金信号”。这些应符合业务和利益相关者的要求。计算 SLIs SLOs、和 SLAs 使用 Amazon CloudWatch 和 Metric Math 对于管理服务可靠性至关重要。遵循本指南中概述的最佳实践，以有效监控和维护 EKS 环境的性能。
3. 然后继续浏览基础设施的不同层，将 EKS 集群、节点和其他基础设施指标与您的工作负载 KPIs关联起来。将您的业务指标和运营指标存储在系统中，您可以将它们关联在一起，并根据观察到的对两者的影响得出结论。
4. EKS 公开来自控制平面、集群 kube-state-metrics、Pod 和节点的指标。所有这些指标的相关性取决于您的需求，但是您可能不需要跨不同层次的每一个指标。您可以使用此 EKS 基本指标指南作为监控 EKS 集群和工作负载整体运行状况的基准。

以下是 prometheus 抓`relabel_config`取配置的示例，我们在其中使用仅保留 kubelet 指标并`metric_relabel_config`删除所有容器指标。

```
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - kube-system
  bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  tls_config:
    insecure_skip_verify: true
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_label_k8s_app]
    regex: kubelet
    action: keep

  metric_relabel_configs:
  - source_labels: [__name__]
    regex: container_(network_tcp_usage_total|network_udp_usage_total|tasks_state|
cpu_load_average_10s)
    action: drop
```

## 在适用的情况下减少基数

基数是指特定指标集的数据值与其维度（例如 prometheus 标签）相结合的唯一性。高基数指标具有多个维度，每个维度指标组合具有更高的唯一性。更高的基数会导致更大的公制遥测数据大小和存储需求，从而增加成本。

在下面的高基数示例中，我们看到指标 “延迟” 具有维度、requestID、CustomerID 和服务，每个维度都有许多唯一的值。基数是衡量每个维度可能值的数量组合的指标。在 Prometheus 中，每组唯一的维度/标签都被视为一个新的指标，因此高基数意味着更多的指标。

在每个指标（集群、命名空间、服务、Pod、容器等）都有许多指标和维度/标签的 EKS 环境中，基数往往会增长。为了优化成本，请仔细考虑您正在收集的指标的基数。例如，如果您要在集群级别聚合用于可视化的特定指标，则可以删除位于较低层的其他标签，例如命名空间标签。

为了识别 prometheus 中的高基数指标，您可以运行以下 PROMQL 查询来确定哪些抓取目标的指标数量最多（基数）：

```
topk_max(5, max_over_time(scrape_samples_scraped[1h]))
```

以下 PROMQL 查询可以帮助您确定哪些抓取目标的指标流失率最高（在给定的抓取中创建了多少新指标系列）：

```
topk_max(5, max_over_time(scrape_series_added[1h]))
```

如果你使用的是 grafana，你可以使用 Grafana Lab 的 Mimirtool 来分析你的 grafana 仪表板，使用 prometheus 规则来识别未使用的高基数指标。按照本指南，了解如何使用`mimirtool analyze`和`mimirtool analyze prometheus`命令来识别仪表板中未引用的活动指标。

## 考虑指标粒度

以更高的精度（例如每秒与每分钟）收集指标可能会对收集和存储的遥测数据产生重大影响，从而增加成本。确定合理的抓取或指标收集间隔，在足够大的粒度以查看暂时性问题和足够低的粒度之间取得平衡，从而具有成本效益。降低用于容量规划和更大时间窗分析的指标的粒度。

以下是默认的 AWS Distro for Opentelemetry (ADOT) EKS 插件收集器配置的片段。

> **Important**
>
> 全局 prometheus 抓取间隔设置为 15 秒。此抓取间隔可以增加，从而减少在 prometheus 中收集的指标数据量。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: my-collector-amp

...

config: |
    extensions:
      sigv4auth:
        region: "+++<YOUR_AWS_REGION>+++" service: "aps"+++</YOUR_AWS_REGION>+++

 receivers:
   #
   # Scrape configuration for the Prometheus Receiver
   # This is the same configuration used when Prometheus is installed using the
 community Helm chart
   #
   prometheus:
     config:
       global:   scrape_interval: 15s
         scrape_timeout: 10s
```

## 跟踪

与追踪相关的主要成本源于痕量存储的生成。通过跟踪，目的是收集足够的数据来诊断和了解性能方面。但是，由于 X-Ray 追踪的成本基于转发到 X-Ray 的数据，因此在转发轨迹后删除跟踪不会降低您的成本。让我们回顾一下在维护数据以便进行适当分析的同时降低追踪成本的方法。

### 应用采样规则

默认情况下，X-Ray 采样率是保守的。定义采样规则，您可以在其中控制所收集的数据量。这将提高性能效率，同时降低成本。通过降低采样率，您可以仅从请求中收集工作负载所需的跟踪，同时保持较低的成本结构。

例如，你有一个 java 应用程序，你想调试 1 条有问题的路由的所有请求的跟踪。

通过 SDK 进行配置以从 JSON 文档加载采样规则

```
{
"version": 2,
  "rules": [
```

```
    {
"description": "debug-eks",
      "host": "*",
      "http_method": "PUT",
      "url_path": "/history/*",
      "fixed_target": 0,
      "rate": 1,
      "service_type": "debug-eks"
    }
  ],
  "default": {
"fixed_target": 1,
    "rate": 0.1
  }
}
```

通过控制台

## 使用 AWS Distro 应用尾部采样适用于 OpenTelemetry (ADOT)

ADOT Tail Sampling 允许您控制服务中摄取的迹线量。但是，Tail Sampling 允许您在请求中的所有跨度都完成之后而不是在开始时定义采样策略。这进一步限制了传输到的原始数据量 CloudWatch，从而降低了成本。

例如，如果您对登录页面流量的 1% 和 10% 的付款页面请求进行采样，则可能会在 30 分钟内留下 300 条跟踪记录。使用过滤特定错误的 ADOT Tail Sampling 规则，您可能只剩下 200 条轨迹，从而减少存储的跟踪数量。

```
processors:
  groupbytrace:
    wait_duration: 10s
    num_traces: 300
    tail_sampling:
    decision_wait: 1s # This value should be smaller than wait_duration
    policies:
      - ..... # Applicable policies**
  batch/tracesampling:
    timeout: 0s # No need to wait more since this will happen in previous processors
    send_batch_max_size: 8196 # This will still allow us to limit the size of the
 batches sent to subsequent exporters

service:
  pipelines:
```

```
    traces/tailsampling:
      receivers: [otlp]
      processors: [groupbytrace, tail_sampling, batch/tracesampling]
      exporters: [awsxray]
```

### 利用亚马逊 S3 存储选项

您应该利用 AWS S3 存储桶及其不同的存储类来存储跟踪。在保留期到期之前将跟踪导出到 S3。使用 Amazon S3 生命周期规则将跟踪数据移至符合您要求的存储类别。

例如，如果您的跟踪记录已存在 90 天，Amazon S3 智能分层可以根据您的使用模式自动将数据移至长期存储。如果您需要稍后再参考跟踪记录，则可以使用 Ama zon Athena 来查询 Amazon S3 中的数据。这可以进一步降低分布式跟踪的成本。

## 其他资源：

- 可观测性最佳实践指南
- 最佳实践指标收集
- AWS re: Invent 2022-亚马逊的可观察性最佳实践 () COP343
- AWS re: Invent 2022-可观察性：现代应用程序的最佳实践 () COP344

# Windows 最佳实践

本指南提供了有关运行 Windows 容器和节点的建议。

主题


- 亚马逊 EKS 优化了 Windows AMI 管理
- 为 Windows 容器和容器配置 GMSA
- Windows 工作节点强化
- 容器镜像扫描
- Windows 服务器版本和许可证
- 日志记录
- 监控
- Windows 联网
- 避免 OOM 错误
- 修补 Windows 服务器和容器
- 运行异构工作负载
- Pod 安全上下文
- 永久存储选项
- 强化 Windows 容器镜像


## 亚马逊 EKS 优化了 Windows AMI 管理

Windows Amazon EKS 优化版 AMI 基于 Windows Server 2019 和 Windows Server 2022 构建。它们被配置作为 Amazon EKS 节点的基本映像。默认情况下，AMI 包括以下组件：

- kubelet
- kube-proxy
- 适用于 Kubernetes 的 AWS IAM Authenticator
- csi-proxy
- containerd

您可以通过查询 AWS Systems Manager Parameter Store API，以编程方式检索亚马逊 EKS 优化的 AMI 的亚马逊系统映像 (AMI) ID。此参数使您无需手动查找 Amazon EKS 优化版 AMI ID。有关 Systems Manager 参数存储 API 的更多信息，请参阅GetParameter。您的用户账户必须具有 ssm:GetParameter IAM 权限才能检索 Amazon EKS 优化的 AMI 元数据。

以下示例检索适用于 Windows Server 2019 LTSC Core 的最新亚马逊 EKS 优化版 AMI 的 AMI 的 AMI ID。AMI 名称中列出的版本号与其准备的相应的 Kubernetes 版本有关。

```
aws ssm get-parameter --name /aws/service/ami-windows-latest/Windows_Server-2019-
English-Core-EKS_Optimized-1.21/image_id --region us-east-1 --query "Parameter.Value"
 --output text
```

输出示例：

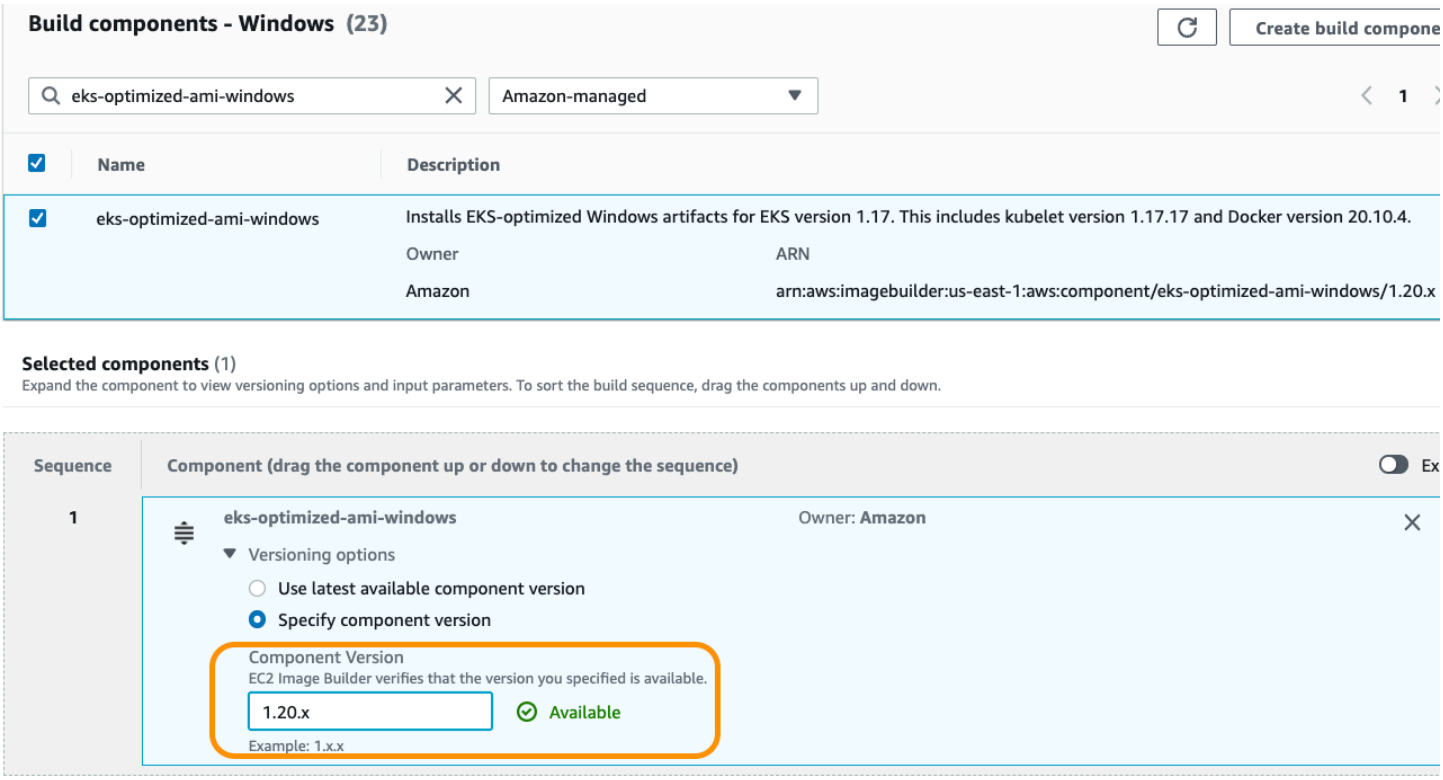
```
ami-09770b3eec4552d4e
```

## 管理你自己的亚马逊 EKS 优化的 Windows AMI

迈向生产环境的一个重要步骤是在整个亚马逊 EKS 集群中保持相同的亚马逊 EKS 优化版 Windows AMI 和 kubelet 版本。

在 Amazon EKS 集群中使用相同的版本可以缩短故障排除时间并提高集群一致性。Amazon EC2 Image Builder 可帮助创建和维护定制的亚马逊 EKS 优化 Windows AMI，以便在亚马逊 EKS 集群中使用。

使用 Amazon EC2 Image Builder 在 Windows 服务器版本、AWS Windows Server AMI 发布日期、and/or 操作系统版本之间进行选择。构建组件步骤允许你在现有的 EKS 优化的 Windows Artifacts 和 kubelet 版本之间进行选择。欲了解更多信息：https://docs.aws.amazon.com/eks/latest/userguide/eks-custom-ami-windows.html




注意：在选择基础映像之前，请查阅 Windows Server 版本和许可证部分，了解有关发布渠道更新的重要详细信息。

## 为自定义 EKS 优化的 AMI 配置更快的启动

使用经过优化 Windows Amazon EKS 的自定义 AMI 时，启用快速启动功能可以将 Windows 工作节点的启动速度提高多达 65%。此功能保留了一组预先配置的快照，这些快照具有 Sysprep 专用、Windows 开箱即用体验 (OOBE) 步骤和已完成所需的重启。这些快照随后用于后续启动，从而缩短了向外扩展或替换节点的时间。只能通过 EC2 控制台或 AWS CLI 为您拥有的 AMI 启用快速启动，并且可以配置所维护的快照数量。

注意：Fast Launch 与 Amazon-provided EKS 优化的默认 AMI 不兼容，请在尝试启用之前按上述方式创建自定义 AMI。

欲了解更多信息：AWS Windows AMI-配置您的 AMI 以加快启动速度

## 在自定义 AMI 上缓存 Windows 基础图层

Windows 容器镜像比 Linux 容器镜像大。如果您正在运行任何容器化的.NET Framework-based 应用程序，则平均图像大小约为 8.24GB。在 Pod 调度期间，必须在 Pod 处于 Running 状态之前将容器镜像完全拉取并提取到磁盘中。

在此过程中，容器运行时 (containerd) 将整个容器镜像提取并提取到磁盘中。拉取操作是一个并行进程，这意味着容器运行时会并行拉取容器镜像层。相比之下，提取操作是按顺序进行的，而且是 I/O 密集型的。因此，容器镜像可能需要 8 分钟以上的时间才能完全提取并可供容器运行时 (containerd) 使用，因此，pod 的启动时间可能需要几分钟。

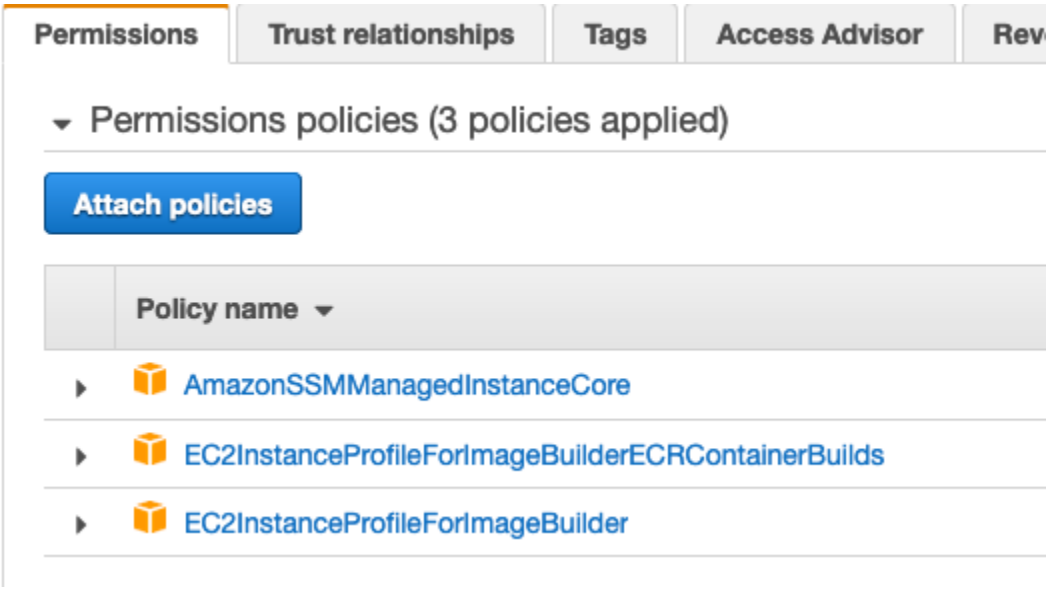
如修补 Windows 服务器和容器主题中所述，可以选择使用 EKS 构建自定义 AMI。在 AMI 准备过程中，您可以添加额外的 EC2 映像生成器组件，以便在本地提取所有必要的 Windows 容器映像，然后生成 AMI。这种策略将大大缩短 pod 变为 “正在运行” 状态的时间。

在 Amazon EC2 Image Builder 上，创建一个组件来下载必要的图像并将其附加到图像配方中。以下示例从 ECR 存储库中提取特定映像。

```
name: ContainerdPull
description: This component pulls the necessary containers images for a cache strategy.
schemaVersion: 1.0

phases:
  - name: build
    steps:
      - name: containerdpull
        action: ExecutePowerShell
        inputs:
          commands:
            - Set-ExecutionPolicy Unrestricted -Force
            - (Get-ECRLoginCommand).Password | docker login --username AWS --password-
stdin 111000111000.dkr.ecr.us-east-1.amazonaws.com
            - ctr image pull mcr.microsoft.com/dotnet/framework/aspnet:latest
            - ctr image pull 111000111000.dkr.ecr.us-east-1.amazonaws.com/
myappcontainerimage:latest
```

为确保以下组件按预期运行，请检查 EC2 Image builder (EC2InstanceProfileForImageBuilder) 使用的 IAM 角色是否附加了策略：




### 博客文章

在以下博客文章中，您将逐步了解如何为自定义 Amazon EKS Windows AMI 实施缓存策略：

使用 EC2 映像生成器和映像缓存策略加快 Windows 容器的启动时间

## 为 Windows 容器和容器配置 GMSA

### 什么是 GMSA 账户

Windows-based 诸如.NET 应用程序之类的应用程序通常使用 Active Directory 作为身份提供者，authorization/authentication 使用 NTLM 或 Kerberos 协议提供。

与 Active Directory 交换 Kerberos 票证的应用程序服务器需要加入域。Windows 容器不支持域加入，也没有多大意义，因为容器是临时资源，会给 Active Directory RID 池带来负担。

但是，管理员可以利用 GMSA Activ e Directory 帐户来协商 Windows 容器、NLB 和服务器群等资源的 Windows 身份验证。

### Windows 容器和 GMSA 用例

利用 Windows 身份验证并作为 Windows 容器运行的应用程序可以从 GMSA 中受益，因为 Windows 节点用于代表容器交换 Kerberos 票证。有两个选项可用于设置 Windows 工作节点以支持 GMSA 集成：

## 1- Domain-joined Windows 工作节点

在此设置中，Windows 工作节点在 Active Directory 域中加入域，Windows 工作节点的 AD 计算机帐户用于针对 Active Directory 进行身份验证并检索用于容器的 GMSA 身份。

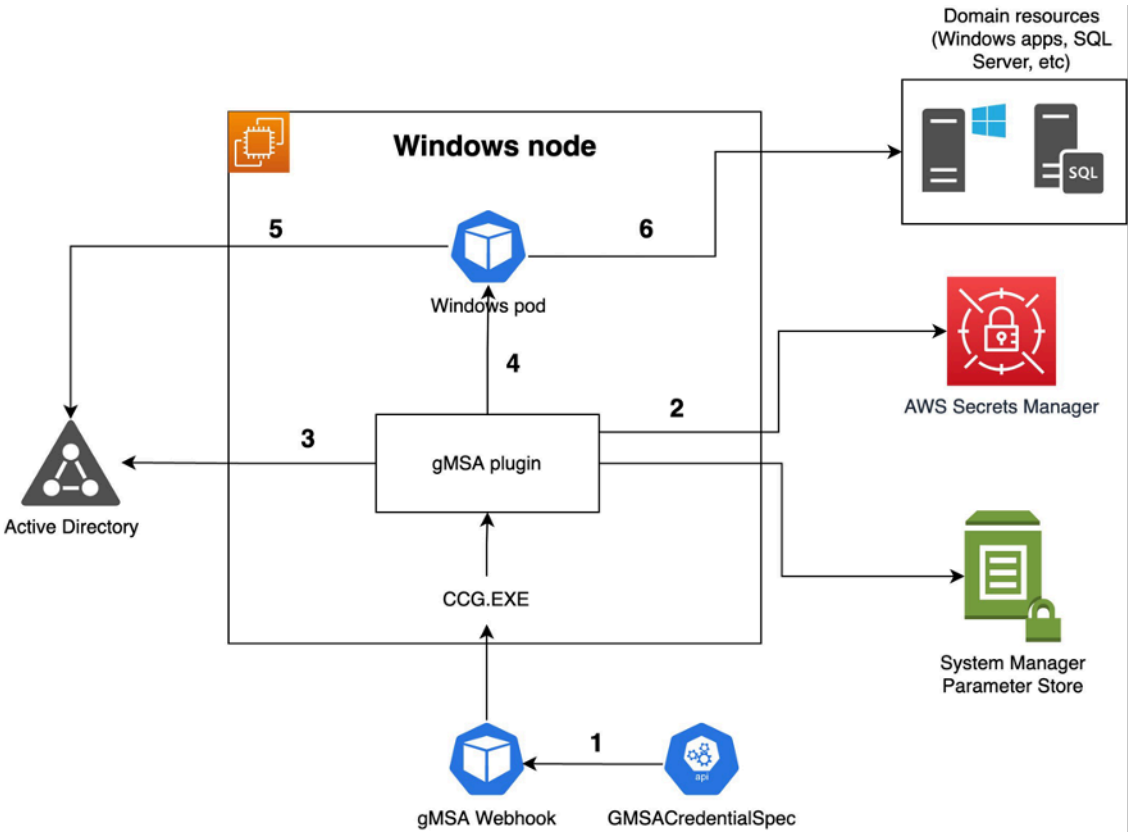
在加入域的方法中，您可以使用现有的 Active Directory GPO 轻松管理和强化 Windows 工作节点；但是，在 Windows 工作节点加入 Kubernetes 集群期间，它会产生额外的操作开销和延迟，因为它需要在节点启动期间额外重新启动，并且在 Kubernetes 集群终止节点后需要清理 Active Directory 车库。

在以下博客文章中，您将找到有关如何实现 Domain-joined Windows 工作节点方法的详细分步说明：

[亚马逊 EKS Windows pod 上的 Windows 身份验证]

## 2-无域 Windows 工作线程节点

在此设置中，Windows 工作节点未加入 Active Directory 域，并使用 “便携式” 身份 (user/password) 对 Active Directory 进行身份验证并检索要用于容器的 GMSA 身份。




便携身份是 Active Directory 用户；身份 (user/password) 存储在 AWS Secrets Manager 或 AWS System Manager 参数存储中，一个 AWS-developed 名为 ccg_plugin 的插件将用于从 AWS Secrets Manager 或 AWS System Manager 参数存储中检索此身份，并将其传递给 containerd 以检索 GMSA 身份并将其提供给容器。

在这种无域方法中，使用 GMSA 时，在 Windows 工作节点启动期间不进行任何活动目录交互，可以减少活动目录管理员的操作开销，从而使您受益匪浅。

在以下博客文章中，您将详细了解如何实现无域 Windows 工作节点方法：

适用于亚马逊 EKS Windows pod 的无域 Windows 身份验证

### 重要提示

尽管 pod 能够使用 GMSA 帐户，但还需要相应地设置应用程序或服务以支持 Windows 身份验证，例如，为了将 Microsoft IIS 设置为支持 Windows 身份验证，你应该通过 dockerfile 进行准备：

```
RUN Install-WindowsFeature -Name Web-Windows-Auth -IncludeAllSubFeature
RUN Import-Module WebAdministration; Set-ItemProperty 'IIS:\AppPools\SiteName' -name
 processModel.identityType -value 2
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/anonymousAuthentication' -Name Enabled -Value
 False -PSPath 'IIS:\' -Location 'SiteName'
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/windowsAuthentication' -Name Enabled -Value
 True -PSPath 'IIS:\' -Location 'SiteName'
```

## Windows 工作节点强化

操作系统强化是操作系统配置、修补和删除不必要的软件包的组合，旨在锁定系统并减少攻击面。最佳做法是使用贵公司所需的强化配置来准备自己的 EKS 优化型 Windows AMI。

AWS 每月都会提供一个新的 EKS 优化版 Windows AMI，其中包含最新的 Windows 服务器安全补丁。但是，无论用户使用的是自我管理的还是托管的节点组，用户仍有责任通过应用必要的操作系统配置来强化其 AMI。

Microsoft 提供了一系列工具，例如 Microsoft 安全合规工具包和安全基准，可帮助您根据自己的安全策略需求实现强化。CIS 基准测试也可用，应在适用于生产环境的 Amazon EKS 优化的 Windows AMI 的基础上实施。

### 使用 Windows 服务器核心减少攻击面

Windows Server Core 是一个最低安装选项，作为 EKS 优化的 Windows AMI 的一部分提供。部署 Windows 服务器核心有几个好处。首先，它的磁盘占用空间相对较小，在服务器核心上为6GB，而在

具有台式机体验的Windows服务器上为10GB。其次，由于其代码库和可用的 API 较小，因此它的攻击面较小。

无论亚马逊版本如何，AWS 每月都会为客户提供新的亚马逊 EKS 优化 EKS-supported 版 Windows AMI，其中包含最新的微软安全补丁。作为最佳实践，Windows 工作节点必须替换为基于最新 Amazon EKS-optimized AMI 的新工作节点。任何运行超过 45 天但未进行更新或节点更换的节点都缺乏安全最佳实践。

## 避开 RDP 连接

远程桌面协议 (RDP) 是微软开发的一种连接协议，旨在为用户提供通过网络连接到另一台 Windows 计算机的图形界面。

作为最佳实践，您应该将 Windows 工作节点当作临时主机对待。这意味着没有管理连接，没有更新，也没有故障排除。任何修改和更新都应作为新的自定义 AMI 实现，并通过更新 Auto Scaling 组来取而代之。请参阅修补 Windows 服务器和容器以及亚马逊 EKS 优化的 Windows AMI 管理。

在部署期间，通过在 ssh 属性上传递值 false 来禁用 Windows 节点上的 RDP 连接，如下例所示：

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

如果需要访问 Windows 节点，请使用 AWS 系统管理器会话管理器通过 AWS 控制台和 SSM 代理建立安全 PowerShell 会话。要了解如何实施该解决方案，请观看使用 AWS Systems Manager 会话管理器安全访问 Windows 实例

要使用系统管理器会话管理器，必须对用于启动 Windows 工作节点的 IAM 角色应用额外的 IAM 策略。以下是在集`eksctl`群清单中指定AmazonSSMManagedInstanceCore的：

```
 nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
```

```
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
  iam:
    attachPolicyARNs:
      - arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
      - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
      - arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
      - arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly
      - arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore
```

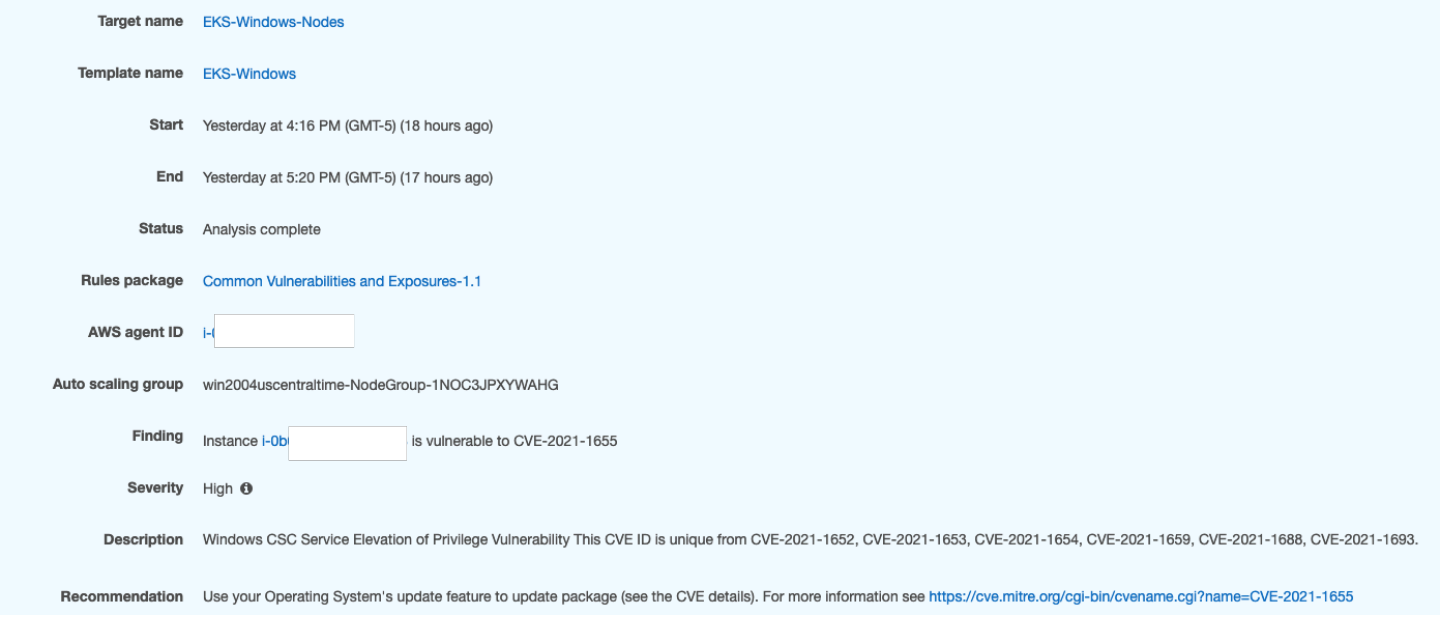
## Amazon Inspector

> Amazon Inspector 是一项自动安全评估服务，可帮助提高部署在 AWS 上的应用程序的安全性和合规性。Amazon Inspector 会自动评估应用程序的暴露程度、漏洞以及与最佳实践的偏差。执行评估后，Amazon Inspector 将生成按严重性级别优先排序的安全调查发现详细列表。这些发现可以直接查看，也可以作为详细评估报告的一部分进行审查，这些报告可通过 Amazon Inspector 控制台或 API 获得。

Amazon Inspector 可用于在 Windows 工作节点上运行 CIS 基准评估，也可以通过执行以下任务将其安装在 Windows 服务器核心上：

1. 下载以下 .exe 文件：https://inspector-agent.amazonaws.com/windows/installer/latest/AWSAgentInstall.exe
2. 将代理转移到 Windows 工作节点上。
3. 运行以下命令 PowerShell 以安装 Amazon Inspector 代理：`.\AWSAgentInstall.exe /install`

以下是第一次运行后的输出。如您所见，它根据 CVE 数据库生成了调查结果。您可以使用它来强化您的工作节点或基于强化配置创建 AMI。




有关 Amazon Inspector 的更多信息，包括如何安装亚马逊 Inspector 代理、设置 CIS 基准评估以及生成报告，请观看使用亚马逊 Inspector 提高 Windows 工作负载的安全性和合规性视频。

## Amazon GuardDuty

> Amazon GuardDuty 是一项威胁检测服务，可持续监控恶意活动和未经授权的行为，以保护您的 AWS 账户、工作负载和存储在 Amazon S3 中的数据。使用云可以简化账户和网络活动的收集和汇总，但是安全团队持续分析事件日志数据中是否存在潜在威胁可能会很耗时。

通过使用 Amazon， GuardDuty 您可以查看针对 Windows 工作节点的恶意活动，例如 RDP 暴力攻击和端口探测攻击。

观看使用亚马逊针对 Windows 工作负载进行威胁检测 GuardDuty视频，了解如何在优化的 EKS Windows AMI 上实施和运行 CIS 基准测试

## 适用于 Windows 的 Amazon EC2 中的安全

阅读适用于 Amazon EC2 Windows 实例的安全最佳实践，以便在每一层实施安全控制。

# 容器镜像扫描

Image Scanning 是一项自动漏洞评估功能，通过扫描应用程序容器映像中是否存在各种操作系统漏洞，有助于提高应用程序容器映像的安全性。

目前，亚马逊弹性容器注册表 (ECR) 只能扫描 Linux 容器镜像中是否存在漏洞。但是，有些第三方工具可以与现有的 Windows 容器映像扫描 CI/CD 管道集成。

- 锚
- PaloAlto 棱镜云
- 趋势科技-趋势科技服务器深度安全防护系统智能检查

要详细了解如何将这些解决方案与 Amazon 弹性容器存储库 (ECR) 集成，请查看：

- Anchore，扫描亚马逊弹性容器注册表 (ECR) 上的图像
- PaloAlto，扫描 Amazon 弹性容器注册表 (ECR) 上的图像
- TrendMicro，扫描 Amazon 弹性容器注册表 (ECR) 上的图像

# Windows 服务器版本和许可证

## Windows Server 版本

亚马逊 EKS 优化的 Windows AMI 基于 Long-Term 服务频道 (LTSC) 上的 Windows Server 2019 和 2022 年数据中心版本。数据中心版本对工作节点上运行的容器数量没有限制。欲了解更多信息：https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/faq

### Long-Term 服务渠道 (LTSC)

以前称为 “Long-Term 服务分支”，这是你已经熟悉的发布模式，每隔 2-3 年就会发布一次 Windows Server 的新主要版本。用户有权获得 5 年的主流支持和 5 年的延期支持。

## 许可

在启动带有 Windows Server-based AMI 的 Amazon EC2 实例时，亚马逊会为您承担许可费用和许可合规性。

# 日志记录

容器化应用程序通常会将应用程序日志定向到 STDOUT。容器运行时会捕获这些日志并对其进行一些处理——通常写入文件。这些文件的存储位置取决于容器的运行时和配置。

与 Windows pod 的一个根本区别是它们不会生成 STDOUT。你可以运行从正在运行LogMonitor的 Windows 容器中检索 ETW（适用于 Windows 的事件跟踪）、Windows 事件日志和其他特定于应用程序的日志，并将格式化的日志输出传输到 STDOUT。然后，可以使用 fluent-bit 或 fluentd 将这些日志流式传输到你想要的目的地，例如亚马逊。 CloudWatch

日志收集机制从 Kubernetes 容器中检索 STDOUT/STDERR 日志。A DaemonSet是从容器收集日志的常用方法。它使您能够独立于应用程序管理日志 routing/filtering /enrichment。fluentd DaemonSet 可用于将这些日志和任何其他应用程序生成的日志流式传输到所需的日志聚合器。

有关从 Windows 工作负载流向的日志流式传输的更多详细信息 CloudWatch ，请参阅此处

## 记录建议

在 Kubernetes 中操作 Windows 工作负载时，一般的日志记录最佳实践没有什么不同。

- 始终记录结构化日志条目 (JSON/SYSLOG)，这样可以更轻松地处理日志条目，因为有许多针对此类结构化格式的预先编写的解析器。
- 集中日志-专用的日志容器专门用于收集来自所有容器的日志消息并将其转发到目的地
- 除非调试，否则请降低日志的详细程度。Verbosity 给日志基础架构带来了很大的压力，重大事件可能会在噪音中丢失。
- 务必将应用程序信息与 transaction/request ID 一起记录下来，以实现可追溯性。Kubernetes 对象不带有应用程序名称，因此例如，在调试日志时，pod 名称`windows-twryrqyw`可能没有任何意义。这有助于使用聚合日志对应用程序进行可追溯性和故障排除。

  如何生成这些 transaction/correlation ID 取决于编程结构。但是一种非常常见的模式是使用日志记录 Aspect/Interceptor，它可以使用 MDC（映射的诊断上下文）为每个传入的请求注入一个唯一的 transaction/correlation ID，如下所示：

```
import org.slf4j.MDC;
import java.util.UUID;
Class LoggingAspect { //interceptor

    @Before(value = "execution(* *.*(..))")
    func before(...) {
        transactionId = generateTransactionId();
        MDC.put(CORRELATION_ID, transactionId);
    }

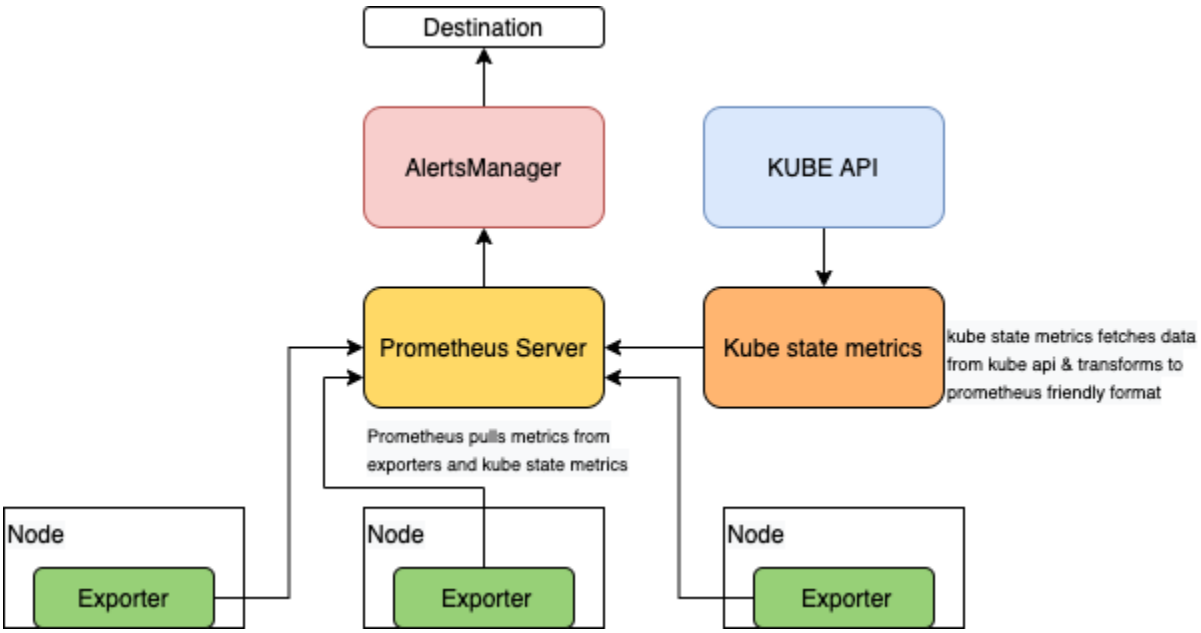
    func generateTransactionId() {
```

```
        return UUID.randomUUID().toString();
    }
}
```

# 监控

Prometheus 是一个 CNCF 毕业项目，是迄今为止最受欢迎的监控系统，可以原生集成到 Kubernetes 中。Prometheus 收集有关容器、容器、节点和集群的指标。此外，Prometheus AlertsManager 利用它可以让你编程警报，在集群中出现问题时发出警告。Prometheus 将指标数据存储为由指标名称和指标对标识的时间序列数据。 key/value Prometheus 包括使用一种名为 PromQL 的语言进行查询的方法，该语言是 Prometheus 查询语言的缩写。

Prometheus 指标收集的高级架构如下所示：




Prometheus 使用拉取机制，使用导出器从目标中抓取指标，并使用 kube 状态指标从 Kubernetes API 中抓取指标。这意味着应用程序和服务必须公开包含 Prometheus 格式指标的 HTTP (S) 端点。然后，Prometheus 将根据其配置，定期从这些 HTTP (S) 端点提取指标。

导出器允许您将第三方指标作为 Prometheus 格式的指标使用。Prometheus 导出器通常部署在每个节点上。有关出口商的完整列表，请参阅 Prometheus 出口商。虽然节点导出器适合导出 Linux 节点的主机硬件和操作系统指标，但它不适用于 Windows 节点。

在带有 Windows 节点的混合节点 EKS 集群中，当你使用稳定的 Prometheus 头盔图表时，你会在 Windows 节点上看到 pod 出现故障，因为这个导出器不适用于 Windows。你需要单独处理 Windows 工作线程池，改为在 Windows 工作节点组上安装 Windows 导出器。

要为 Windows 节点设置 Prometheus 监控，你需要在 Windows 服务器本身上下载并安装 WMI 导出器，然后在 Prometheus 配置文件的抓取配置中设置目标。发布页面提供了所有可用的.msi 安装程序，以及相应的功能集和错误修复。安装程序会将 windows_exporter 设置为 Windows 服务，并在 Windows 防火墙中创建例外。如果安装程序在没有任何参数的情况下运行，则导出器将使用已启用的收集器、端口等的默认设置运行。

你可以查看本指南的调度最佳实践部分，该部分建议使用 taints/tolerations 或有选择地将节点导出器部署 RuntimeClass 到Linux节点，而Windows导出器是在你引导节点时安装在Windows节点上的，或者使用你选择的配置管理工具（例如chef、Ansible、SSM等）。

请注意，与节点导出器作为守护程序安装的 Linux 节点不同，在 Windows 节点上，WMI 导出器安装在主机本身上。导出器将导出 CPU 使用率、内存和磁盘 I/O 使用率等指标，还可用于监控 IIS 站点和应用程序、网络接口和服务。

默认情况下，Windows_exporter 将显示来自已启用收集器的所有指标。这是收集指标以避免错误的推荐方法。但是，对于高级用途，可以向 windows_exporter 传递一个可选的收集器列表来筛选指标。Prometheus 配置中的 collect [] 参数可以让你做到这一点。

Windows 的默认安装步骤包括在引导过程中将导出器作为服务下载并启动，并使用参数（例如要筛选的收集器）。

```
> Powershell Invoke-WebRequest https://github.com/prometheus-community/
windows_exporter/releases/download/v0.13.0/windows_exporter-0.13.0-amd64.msi -OutFile
 <DOWNLOADPATH>

> msiexec /i <DOWNLOADPATH>
 ENABLED_COLLECTORS="cpu,cs,logical_disk,net,os,system,container,memory"
```

默认情况下，可以在端口 9182 上的 /metrics 端点抓取指标。此时，Prometheus 可以通过在 Prometheus 配置中添加以下 scrape_config 来使用这些指标

```
scrape_configs:
    - job_name: "prometheus"
      static_configs:
        - targets: ['localhost:9090']
    ...
    - job_name: "wmi_exporter"
      scrape_interval: 10s
      static_configs:
        - targets: ['<windows-node1-ip>:9182', '<windows-node2-ip>:9182', ...]
```

Prometheus 配置使用重新加载

```
> ps aux | grep prometheus
> kill HUP <PID>
```

添加目标的更好、推荐的方法是使用名为的自定义资源定义 ServiceMonitor，该定义是 Prometheus 运算符] 的一部分，它为对象和控制器提供定义，控制器将激活 ServiceMonitors 我们定义的并自动生成所需的 Prometheus 配置。 ServiceMonitor

声明性地指定了应如何监控 Kubernetes 服务组，用于定义您希望从 Kubernetes 中获取指标的应用程序。 ServiceMonitor在其中 ServiceMonitor 我们指定了 Kubernetes 标签，操作员可以使用这些标签来识别 Kubernetes 服务，该服务反过来识别我们想要监控的 Pod。

为了利用 ServiceMonitor，请创建一个指向特定 Windows 目标的 Endpoint 对象、一个无头服务和一个 ServiceMontor 用于 Windows 节点的无头服务。

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
subsets:
- addresses:
  - ip: NODE-ONE-IP
    targetRef:
      kind: Node
      name: NODE-ONE-NAME
  - ip: NODE-TWO-IP
    targetRef:
      kind: Node
      name: NODE-TWO-NAME
  - ip: NODE-THREE-IP
    targetRef:
      kind: Node
      name: NODE-THREE-NAME
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP
```

```
---
apiVersion: v1
kind: Service ##Headless Service
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
spec:
  clusterIP: None
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP
    targetPort: 9182
  sessionAffinity: None
  type: ClusterIP

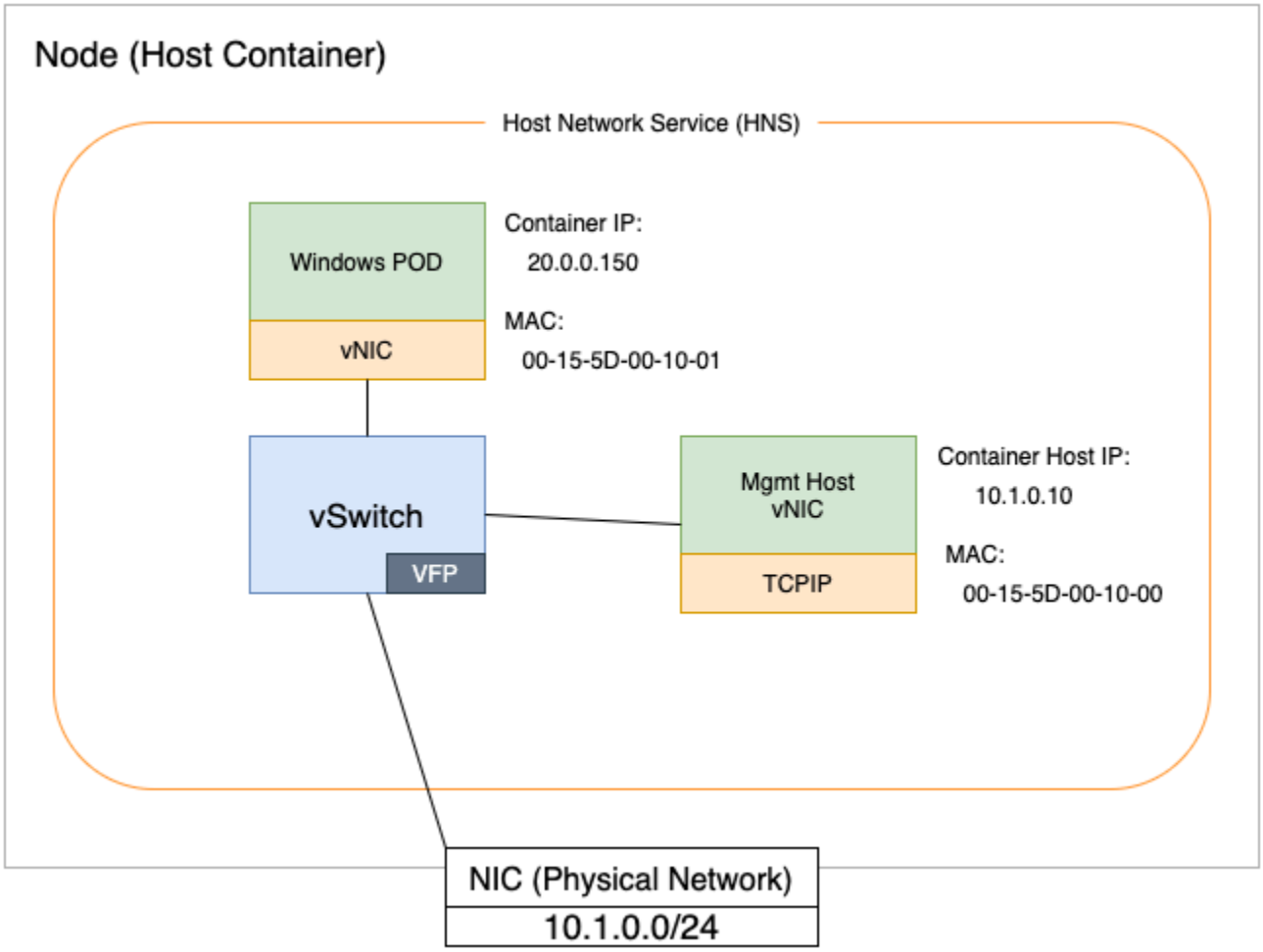
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor ##Custom ServiceMonitor Object
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: monitoring
spec:
  endpoints:
  - interval: 30s
    port: http-metrics
  jobLabel: k8s-app
  namespaceSelector:
    matchNames:
    - kube-system
  selector:
    matchLabels:
      k8s-app: wmiexporter
```

有关操作员和用法的更多详细信息 ServiceMonitor，请查看官方运营商文档。请注意，Prometheus 确实支持使用许多服务发现选项进行动态目标发现。

# Windows 联网

## Windows 容器网络概述

Windows 容器与 Linux 容器有根本的不同。Linux 容器使用 Linux 结构，例如命名空间、联合文件系统和 cgroups。在 Windows 上，这些结构是由主机计算服务 (HCS) 从 containerd 中抽象出来的。HCS 充当一个 API 层，位于 Windows 上的容器实现之上。Windows 容器还利用主机网络服务 (HNS) 来定义节点上的网络拓扑。




从网络的角度来看，HCS 和 HNS 使 Windows 容器的功能类似于虚拟机。例如，每个容器都有一个虚拟网络适配器 (vNIC)，该适配器连接到 Hyper-V 虚拟交换机 (vSwitch)，如上图所示。

# IP 地址管理

Amazon EKS 中的一个节点使用其弹性网络接口 (ENI) 连接到 AWS VPC 网络。目前，每个 Windows 工作节点仅支持一个 ENI。Windows 节点的 IP 地址管理由在控制平面中运行的 VPC 资源控制器执行。有关 Windows 节点 IP 地址管理工作流程的更多详细信息，可在此处找到。

Windows 工作节点可以支持的 pod 数量取决于节点的大小和可用 IPv4 地址的数量。您可以计算节点上可用的 IPv4 地址，如下所示：

- 默认情况下，只为弹性网卡分配辅助 IPv4 地址。在这种情况下：

```
Total IPv4 addresses available for Pods = Number of supported IPv4 addresses in the
 primary interface - 1
```

  我们从总数中减去一个，因为一个 IPv4 地址将用作 ENI 的主地址，因此无法分配给 Pod。

- 如果已通过启用前缀委派功能将集群配置为高 pod 密度，那么-

```
Total IPv4 addresses available for Pods = (Number of supported IPv4 addresses in the
 primary interface - 1) * 16
```

  在这里，VPC 资源控制器将分配辅助 IPv4 地址，而不是分配/28 `prefixes`辅助 IPv4 地址，因此，可用 IPv4 地址的总数将增加 16 倍。

使用上面的公式，我们可以计算基于 m5.large 实例的 Windows 工作节点的最大 pod 数，如下所示：

- 默认情况下，在辅助 IP 模式下运行时-

```
10 secondary IPv4 addresses per ENI - 1 = 9 available IPv4 addresses
```

- 使用时 `prefix delegation`-

```
(10 secondary IPv4 addresses per ENI - 1) * 16 = 144 available IPv4 addresses
```

有关实例类型可以支持多少 IP 地址的更多信息，请参阅每种实例类型的每个网络接口的 IP 地址。

另一个关键考虑因素是网络流量。使用 Windows 时，在拥有超过 100 个服务的节点上存在端口耗尽的风险。出现这种情况时，节点将开始抛出错误，并显示以下消息：

“策略创建失败：Win32 中的 hcn CreateLoadBalancer 失败：指定的端口已经存在。”

为了解决这个问题，我们利用服务器直接返回 (DSR)。DSR 是一种非对称网络负载分配的实现。换句话说，请求和响应流量使用不同的网络路径。此功能可加快 Pod 之间的通信速度并降低端口耗尽的风险。因此，我们建议在 Windows 节点上启用 DSR。

在 Windows Server SAC EKS 优化的 AMI 中，DSR 默认处于启用状态。对于 Windows Server 2019 LTSC EKS 优化的 AMI，你需要使用下面的脚本在实例配置期间启用它，并在 NodeGroup 中使用 Windows Server 2019 Full 或 Core 作为 amiFamily。`eksctl`有关更多信息，请参阅 eksctl 自定义 AMI。

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

为了在 Windows Server 2019 及更高版本中使用 DSR，你需要在实例启动期间指定以下 kube-proxy 标志。您可以通过调整与自管理节点组启动模板关联的用户数据脚本来做到这一点。

```
<powershell>
[string]$EKSBinDir = "$env:ProgramFiles\Amazon\EKS"
[string]$EKSBootstrapScriptName = 'Start-EKSBootstrap.ps1'
[string]$EKSBootstrapScriptFile = "$EKSBinDir\$EKSBootstrapScriptName"
(Get-Content $EKSBootstrapScriptFile).replace('"--proxy-mode=kernelspace",', '"--proxy-
mode=kernelspace", "--feature-gates WinDSR=true", "--enable-dsr",') | Set-Content
 $EKSBootstrapScriptFile
& $EKSBootstrapScriptFile -EKSClusterName "eks-windows" -APIServerEndpoint "https://
<REPLACE-EKS-CLUSTER-CONFIG-API-SERVER>" -Base64ClusterCA "<REPLACE-EKSCLUSTER-
CONFIG-DETAILS-CA>" -DNSClusterIP "172.20.0.10" -KubeletExtraArgs "--node-
labels=alpha.eksctl.io/cluster-name=eks-windows,alpha.eksctl.io/nodegroup-name=windows-
ng-ltsc2019 --register-with-taints=" 3>&1 4>&1 5>&1 6>&1
</powershell>
```

可以按照微软网络博客和 AWS 实验室上的 Windows 容器中的说明验证 DSR 的启用。

```
PS C:\temp> (Get-Content C:\temp\pmsxleau.0ia\policy.txt | ConvertFrom-JSON).Policies


ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 17
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 443
InternalPort : 443
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.1}
```

如果保留可用的 IPv4 地址并最大限度地减少浪费对您的子网至关重要，那么通常建议避免使用 Windows 前缀模式-何时应避免使用前缀委派模式。如果仍需要使用前缀委派，则可以采取措施优化子网中的 IPv4 地址利用率。有关如何微调 IPv4 地址请求和分配过程的详细说明，请参阅配置前缀委派参数。调整这些配置可以帮助您在节省 IPv4 地址和前缀委派的 pod 密度优势之间取得平衡。

使用分配辅助 IPv4 地址的默认设置时，目前不支持用于操作 VPC 资源控制器如何请求和分配 IPv4 地址的配置。更具体地说`minimum-ip-target`，`warm-ip-target`只有前缀委托模式才支持和。另请注意，在辅助 IP 模式下，根据接口上的可用 IP 地址，VPC 资源控制器通常会代表您在节点上分配 3 个未使用的 IPv4 地址，以维护热 IP，从而缩短 Pod 启动时间。如果您想最大限度地减少未使用的热 IP 地址的 IP 浪费，则可以设法在给定的 Windows 节点上安排更多 pod，以便尽可能多地使用 ENI 的 IP 地址容量。更明确地说，如果节点和正在运行的 pod 都已在使用 ENI 上的所有 IP 地址，则可以避免对未使用的 IP 进行预热。另一种帮助您解决子网 IP 地址可用性限制的解决方法是探索增加子网大小或将 Windows 节点分成自己的专用子网。

此外，值得注意的是，目前 Windows 节点不支持 IPv6。

## 容器网络接口 (CNI) 选项

AWSVPC CNI 实际上是适用于 Windows 和 Linux 工作节点的 CNI 插件。尽管 AWSVPC CNI 可以满足许多客户的需求，但有时您仍需要考虑诸如覆盖网络之类的替代方案，以避免 IP 耗尽。在这些情况下，可以使用 Calico CNI 代替 AWSVPC CNI。P@@ roject Calico 是由 Tig era 开发的开源软件。该

软件包括可与EKS配合使用的CNI。在 EKS 中安装 Calico CNI 的说明可以在 P roject Calico EKS 安装页面上找到。

## 网络策略

从 Kubernetes 集群上的 Pod 之间的默认开放通信模式更改为基于网络策略限制访问的默认模式，这被认为是一种最佳实践。开源 P roject Calico 为同时适用于 Linux 和 Windows 节点的网络策略提供了强有力的支持。此功能是独立的，不依赖于使用 Calico CNI。因此，我们建议安装 Calico 并将其用于网络策略管理。

在 EKS 中安装 Calico 的说明可以在亚马逊 EKS 上安装 Calico 页面上找到。

此外，《Amazon EKS 安全最佳实践指南-网络》部分中提供的建议同样适用于带有 Windows 工作节点的 EKS 集群，但是，Windows 目前不支持某些功能，例如 “Pod 的安全组”。

# 避免 OOM 错误

Windows 不像 Linux 那样有内存不足的进程杀手。Windows 始终将所有用户模式的内存分配视为虚拟内存分配，并且页面文件是必需的。最终效果是，Windows不会像Linux那样达到内存不足的情况。进程将分页到磁盘，而不是被内存不足 (OOM) 终止。如果内存过度配置且所有物理内存耗尽，则分页可能会降低性能。

## 预留系统和 kubelet 内存

与 Linux 不同，后者为 kubernetes 系统守护程序（例如 kubelet、容器运行时等）`--kubelet-reserve`捕获资源预留；为 sshd、udev 等操作系统守护程序`--system-reserve`捕获资源预留。在 Windows 上，这些标志不会捕获和设置 kubelet 或节点上运行的进程的内存限制。

但是，你可以将这些标志组合起来，设法NodeAllocatable减少节点的容量，同时使用 Pod 清单内存资源限制来控制每个 Pod 的内存分配。使用这种策略，您可以更好地控制内存分配，并可以最大限度地减少 Windows 节点上的内存不足 (OOM)。

在 Windows 节点上，最佳做法是为操作系统和进程预留至少 2GB 的内存。`--kubelet-reserve` and/or `--system-reserve`用于减少 NodeAllocatable。

在 Amazon EK Self-managed S Windows 节点文档之后，使用 CloudFormation 模板启动一个新的 Windows 节点组，并对 kubelet 配置进行了自定义。 CloudFormation 有一个名为的元素`BootstrapArguments`，该元素与相同`KubeletExtraArgs`。与以下标志和值一起使用：

```
--kube-reserved memory=0.5Gi,ephemeral-storage=1Gi --system-
reserved memory=1.5Gi,ephemeral-storage=1Gi --eviction-hard
 memory.available<200Mi,nodefs.available<10%"
```

如果 eksctl 是部署工具，请查看以下文档以自定义 kubelet 配置 https://eksctl.io/usage/customizing-the-kubelet/

## Windows 容器内存要求

根据微软的文档，适用于NANO的Windows Server基础映像至少需要30MB，而服务器核心需要45MB。随着你添加 Windows 组件（例如.NET 框架、作为 IIS 的 Web 服务）和应用程序，这些数字会增加。

你必须知道 Windows 容器镜像（即基础镜像及其应用程序层）所需的最低内存量，并将其设置为 pod 规范 resources/requests 中的容器。你还应该设置一个限制，以避免 pod 在出现应用程序问题时消耗所有可用的节点内存。

在下面的示例中，当 Kubernetes 调度器尝试在节点上放置 Pod 时，Pod 的请求用于确定哪个节点有足够的资源可供调度。

```
spec:
 - name: iis
   image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
   resources:
     limits:
       cpu: 1
       memory: 800Mi
     requests:
       cpu: .1
       memory: 128Mi
```

## 结论

使用这种方法可以最大限度地降低内存耗尽的风险，但不能防止内存耗尽的发生。使用 Amazon CloudWatch Metrics，您可以设置警报和补救措施，以防出现内存耗尽的情况。

# 修补 Windows 服务器和容器

修补 Windows 服务器是 Windows 管理员的一项标准管理任务。这可以使用不同的工具来完成，例如 Amazon System Manager-补丁管理器、WSUS、系统中心配置管理器等。但是，不应将亚马逊 EKS

集群中的 Windows 节点视为普通的 Windows 服务器。它们应该被视为不可变的服务器。简而言之，避免更新现有节点，只需基于新更新的 AMI 启动一个新节点即可。

使用 EC2 Image Builder，您可以通过创建配方和添加组件来自动构建 AMI。

以下示例显示了组件，这些组件可以是由 AWS (Amazon-managed) 构建的预先存在的组件，也可以是您创建的组件（归我所有）。请密切注意名为 update-windows 的 Amazon-managed 组件，它会在通过 EC2 Image Builder 管道生成 AMI 之前更新 Windows 服务器。

**Associated components**

Search components

| Component name | Version | Description | Date created | Platform | Owner | ARN |
|---|---|---|---|---|---|---|
| eks-optimized-ami-windows | 1.20.0 | Installs EKS-optimized Windows artifacts for EKS version 1.20. This includes kubelet version 1.20.4 and Docker version 20.10.6. | Aug 18, 2021 6:33 PM | Windows | Amazon | arn:aws:imagebuilder:us-east-1:aws:component/eks-optimized-ami-windows/1.20.0/1 |
| update-windows | 1.0.1 | Updates Windows with the latest security updates. | Aug 18, 2021 6:33 PM | Windows | Amazon | arn:aws:imagebuilder:us-east-1:aws:component/update-windows/1.0.1/1 |
| Docker Pull | 1.0.1 | - | Dec 22, 2021 12:08 PM | Windows | [redacted] | arn:aws:imagebuilder:us-east-[redacted]component/docker-pull/1.0.1/1 |
| simple-boot-test-windows | 1.0.0 | Executes a simple boot test. | Nov 30, 2019 11:39 PM | Windows | Amazon | arn:aws:imagebuilder:us-east-1:aws:component/simple-boot-test-windows/1.0.0/1 |

EC2 Image Builder 允许您基于亚马逊托管公有 AMI 构建 AMI，并对其进行自定义以满足您的业务需求。然后，您可以将这些 AMI 与启动模板相关联，这样您就可以将新的 AMI 关联到 EKS 节点组创建的 Auto Scaling 组。完成后，您可以开始终止现有的 Windows 节点，并且将基于新更新的 AMI 启动新节点。

## 推送和拉取 Windows 镜像

亚马逊发布了 EKS 优化的 AMI，其中包括两个缓存的 Windows 容器镜像。

```
mcr.microsoft.com/windows/servercore
mcr.microsoft.com/windows/nanoserver
```

```
REPOSITORY                                                      TAG                 IMAGE ID       CREATED          SIZE
mcr.microsoft.com/windows/servercore                            ltsc2019            617dd5ead5b8   6 weeks ago      5.74GB
mcr.microsoft.com/windows/nanoserver                            1809                8334af2fa7ba   6 weeks ago      257MB
[redacted].dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest              721afca2c725   7 weeks ago      6.96GB
fluent/fluentd                                                  v1.14-windows-ltsc2019-1   721afca2c725   7 weeks ago      6.96GB
amazonaws.com/eks/pause-windows                                 latest              6392f69ae6e7   10 months ago    255MB
```

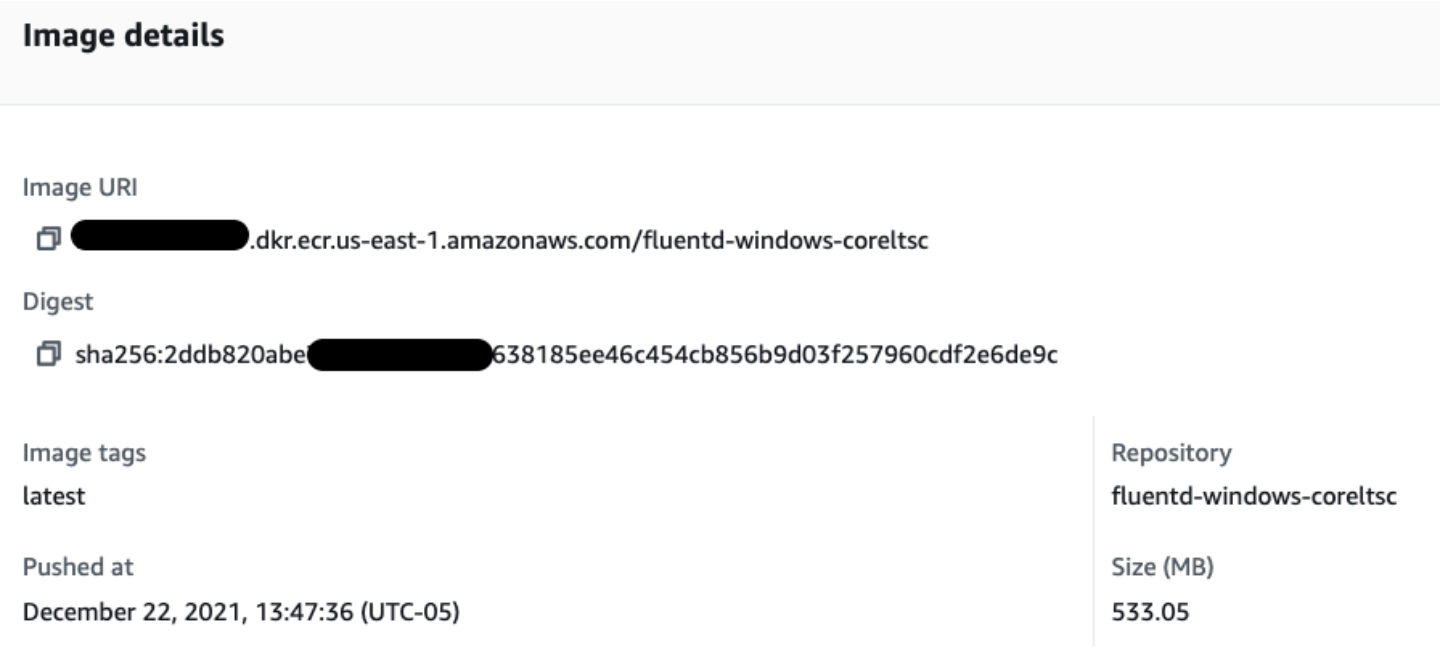
缓存的图像将在主操作系统更新后更新。当微软发布直接影响Windows容器基础映像的新的Windows更新时，该更新将作为普通的Windows更新在主操作系统上启动。使环境保持最新状态可在节点和容器级别提供更安全的环境。

Windows 容器镜像的大小会影响 push/pull 操作，从而导致容器启动时间变慢。缓存 Windows 容器映像允许在创建 AMI 版本而不是容器启动时进行昂贵的 I/O 操作（文件提取）。因此，所有必需的图像

层都将提取到 AMI 上并可供使用，从而加快 Windows 容器启动并开始接受流量的时间。在推送操作期间，只有构成图像的图层才会上传到存储库。

以下示例显示，在亚马逊 ECR 上，fluentd-windows-s ac2004 的图片只有 390.18MB。这是推送操作期间发生的上传量。

以下示例显示了推送到亚马逊 ECR 存储库的 fluentd Windows ltsc 镜像。存储在 ECR 中的图层大小为 533.05 MB。




下面的输出来自 `docker image ls` fluentd v1.14-windows-ltsc2019-1的磁盘大小为 6.96GB，但这并不意味着它下载并提取了那么多的数据。

实际上，在拉取操作期间，只会下载和提取压缩后的533.05MB。

```
REPOSITORY                                                                  TAG
            IMAGE ID       CREATED         SIZE
111122223333.dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest
            721afca2c725   7 weeks ago     6.96GB
fluent/fluentd                                                              v1.14-windows-
ltsc2019-1   721afca2c725   7 weeks ago     6.96GB
amazonaws.com/eks/pause-windows                                             latest
            6392f69ae6e7   10 months ago   255MB
```

大小栏显示图像的总体大小，为 6.96GB。分解一下：

- Windows Server Core 2019 LTSC 基本镜像 = 5.74GB

- Fluentd 未压缩基础镜像 = 6.96GB
- 磁盘上的差异 = 1.2GB
- Fluentd 压缩的最终图像 EC R = 533.05MB

本地磁盘上已存在基础映像，因此磁盘上的总容量额外增加了 1.2GB。下次在 “大小” 列中看到 GB 量时，请不要太担心，磁盘上可能已经有超过 70% 作为缓存的容器镜像。

### 参考

使用 EC2 映像生成器和映像缓存策略加快 Windows 容器的启动时间

## 运行异构工作负载

Kubernetes 支持异构集群，你可以在同一个集群中混合使用 Linux 和 Windows 节点。在该集群中，你可以混合使用在 Linux 上运行的 Pod 和在 Windows 上运行的 Pod。你甚至可以在同一个集群中运行多个版本的 Windows。但是，在做出此决定时，需要考虑几个因素（如下所述）。

### 将 POD 分配给节点最佳实践

为了将 Linux 和 Windows 工作负载保留在各自的 OS-specific 节点上，您需要使用节点选择器和 taints/tolerations的某种组合。在异构环境中调度工作负载的主要目标是避免破坏与现有 Linux 工作负载的兼容性。

### 确保 OS-specific 工作负载落在相应的容器主机上

用户可以确保可以使用 NodeSelectors 在相应的主机上调度 Windows 容器。如今，所有 Kubernetes 节点都有以下默认标签：

```
kubernetes.io/os = [windows|linux]
kubernetes.io/arch = [amd64|arm64|...]
```

如果 Pod 规范不包含类似 NodeSelector 的`"kubernetes.io/os": windows`，则可以将 Pod 调度到任何主机，无论是 Windows 还是 Linux。这可能会出现问题，因为 Windows 容器只能在 Windows 上运行，而 Linux 容器只能在 Linux 上运行。

在企业环境中，有大量预先存在的 Linux 容器部署以及由现成配置（例如 Helm charts）组成的生态系统的情况并不少见。在这些情况下，您可能对更改部署的 NodeSelectors 犹豫不决。另一种方法是使用 Taints。

例如：`--register-with-taints='os=windows:NoSchedule'`

如果你使用的是 EKS，eksctl 提供了通过 ClusterConfig 应用污点的方法：

```
NodeGroups:
  - name: windows-ng
    amiFamily: WindowsServer2022FullContainer
    ...
    labels:
      nodeclass: windows2022
    taints:
      os: "windows:NoSchedule"
```

为所有 Windows 节点添加污点，调度器不会在这些节点上调度 pod，除非它们能容忍污点。Pod 清单示例：

```
nodeSelector:
    kubernetes.io/os: windows
tolerations:
    - key: "os"
      operator: "Equal"
      value: "windows"
      effect: "NoSchedule"
```

## 在同一个集群中处理多个 Windows 构建

每个 Pod 使用的 Windows 容器基础镜像必须与节点相同的内核编译版本相匹配。如果你想在同一个集群中使用多个 Windows Server 版本，那么你应该设置额外的节点标签、NodeSelectors 或者利用一个名为 windows-build 的标签。

Kubernetes 1.17 会自动添加一个新标签 node.kubernetes。 io/windows-build 以简化对同一集群中多个 Windows 版本的管理。如果你运行的是旧版本，则建议手动将此标签添加到 Windows 节点。

此标签反映了兼容性需要匹配的 Windows 主版本、次要版本号和内部版本号。以下是当今每个 Windows 服务器版本使用的值。

值得注意的是，Windows Server 正在转向 Long-Term 服务渠道 (LTSC) 作为主要发布渠道。Windows 服务器 Semi-Annual 频道 (SAC) 已于 2022 年 8 月 9 日停用。Windows Server 将来不会有 future 的 SAC 版本。

| 产品名称 | 内部版本号 |
| --- | --- |
| 服务器已满 2022 LTSC | 10.0.20348 |
| 2019 年服务器核心 LTSC | 10.0.17763 |

可以通过以下命令检查操作系统版本号：

```
kubectl get nodes -o wide
```

KERNEL-VERSION 输出与 Windows 操作系统的内部版本相匹配。

```
NAME                           STATUS   ROLES    AGE   VERSION                INTERNAL-
IP   EXTERNAL-IP     OS-IMAGE                          KERNEL-VERSION
 CONTAINER-RUNTIME
ip-10-10-2-235.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.2.235   3.236.30.157    Windows Server 2022 Datacenter   10.0.20348.1607
         containerd://1.6.6
ip-10-10-31-27.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.31.27   44.204.218.24   Windows Server 2019 Datacenter   10.0.17763.4131
         containerd://1.6.6
ip-10-10-7-54.ec2.internal    Ready    <none>   31m   v1.24.11-eks-a59e1f0   10.10.7.54
    3.227.8.172     Amazon Linux 2                  5.10.173-154.642.amzn2.x86_64
 containerd://1.6.19
```

以下示例将额外的 NodeSelector 应用于 pod 清单，以便在运行不同的 Windows 节点组操作系统 Windows-build 版本时匹配正确的版本。

```
nodeSelector:
    kubernetes.io/os: windows
    node.kubernetes.io/windows-build: '10.0.20348'
tolerations:
    - key: "os"
    operator: "Equal"
    value: "windows"
    effect: "NoSchedule"
```

# Pod 清单中的简 NodeSelector 化和容忍度使用 RuntimeClass

您还可以利用 RuntimeClass 来简化使用污点和容差的过程。这可以通过创建一个用于封装这些污点和容忍度的 RuntimeClass 对象来实现。

RuntimeClass 通过运行以下清单来创建：

```
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
  name: windows-2022
handler: 'docker'
scheduling:
  nodeSelector:
    kubernetes.io/os: 'windows'
    kubernetes.io/arch: 'amd64'
    node.kubernetes.io/windows-build: '10.0.20348'
  tolerations:
  - effect: NoSchedule
    key: os
    operator: Equal
    value: "windows"
```

创建 Runtimeclass 后，在 Pod 清单上将其用作规格进行分配：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis-2022
  labels:
    app: iis-2022
spec:
  replicas: 1
  template:
    metadata:
      name: iis-2022
      labels:
        app: iis-2022
    spec:
      runtimeClassName: windows-2022
      containers:
      - name: iis
```

## 托管节点组 Support

为了帮助客户以更简化的方式运行他们的 Windows 应用程序，AWS 于 2022 年 12 月 15 日启动了对亚马逊 EKS 托管节点组 (MNG) 支持 Windows 容器的支持。为了帮助协调运营团队，使用与 Linux MNG 相同的工作流程和工具启用 Windows MN G。支持 Windows Server 2019 和 2022 的完整版和核心 AMI（亚马逊机器映像）系列版本。

托管节点组 (MNG) 支持以下 AMI 系列。

| AMI 家族 |
| --- |
| Windows_core_2019_x86_64 |
| Windows_full_2019_x86_64 |
| Windows_core_2022_x86_64 |
| Windows_full_2022_x86_64 |

## 其他文档

AWS 官方文档：https://docs.aws.amazon.com/eks/latest/userguide/windows-support.html

要更好地了解 Pod 网络 (CNI) 的工作原理，请查看以下链接：https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html

关于在 EKS 上部署适用于 Windows 的托管节点组的 AWS 博客：https://aws.amazon.com/blogs/containers/deploying-amazon-eks-windows-managed-node-groups/

# Pod 安全上下文

Pod 安全策略 (PSP) 和 P od 安全标准 (PSS) 是在 Kubernetes 中强制执行安全的两种主要方式。请注意，从 Kubernetes v1.21 起， PodSecurityPolicy 它已被弃用，并将在 v1.25 中移除，Pod 安全标准 (PSS) 是 Kubernetes 推荐的未来强制安全性的方法。

Pod 安全策略 (PSP) 是 Kubernetes 中用于实现安全策略的原生解决方案。PSP 是一种集群级别的资源，用于控制 Pod 规范中对安全敏感的方面。使用 Pod 安全策略，您可以定义 Pod 必须满足的一组条件才能被集群接受。PSP 功能从 Kubernetes 的早期就已推出，旨在阻止在给定集群上创建配置错误的 pod。

有关 Pod 安全策略的更多信息，请参阅 Kubernetes 文档。根据 Kubernetes 弃用政策，旧版本将在该功能弃用九个月后停止获得支持。

另一方面，Pod 安全标准 (PSS) 是推荐的安全方法，通常使用安全上下文实现，被定义为 Pod 清单中 Pod 和容器规范的一部分。PSS 是 Kubernetes 项目团队为解决 Pods 安全相关的最佳实践而定义的官方标准。它定义了诸如基准（限制性最低，默认）、特权（非限制）和限制（最严格）之类的策略。

我们建议从基准配置文件开始。PSS 基准配置文件在安全性和潜在摩擦之间取得了稳固的平衡，需要最少的例外清单，它是工作负载安全的良好起点。如果您目前正在使用 PSP，我们建议您切换到 PSS。有关 PSS 策略的更多详细信息可以在 Kubernetes 文档中找到。这些政策可以通过多种工具来执行，包括 OPA和 Ky verno的工具。例如，Kyverno在此处提供了PSS策略的完整集合。

安全上下文设置允许用户授予选择进程的权限、使用程序配置文件将功能限制为单个程序、允许权限升级、筛选系统调用等。

在安全上下文方面，Kubernetes 中的 Windows 容器与标准 Linux-based 工作负载有一些限制和区别。

Windows 使用每个容器的 Job 对象和系统命名空间筛选器来包含容器中的所有进程，并提供与主机的逻辑隔离。如果没有命名空间过滤，就无法运行 Windows 容器。这意味着无法在主机的上下文中声明系统权限，因此特权容器在 Windows 上不可用。

以下`windowsOptions`是唯一记录在案的 Windows 安全上下文选项，其余则是常规安全上下文选项

如需查看 Windows 与 linux 中支持的安全上下文属性的列表，请参阅此处的官方文档。

Pod 特定的设置适用于所有容器。如果未指定，则 PodSecurityContext 将使用中的选项。如果同时在 SecurityContext 和中设置 PodSecurityContext，则中指定的值优 SecurityContext 先。

例如，Pod 和容器的运行AsUserName 设置（Windows 选项）与运行设置大致等同于 Linux-specific 运行AsUser 设置，在以下清单中，Pod 特定的安全上下文应用于所有容器

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
```

```
  nodeSelector:
    kubernetes.io/os: windows
```

而在下文中，容器级别的安全上下文会覆盖 pod 级别的安全上下文。

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    ..
    securityContext:
        windowsOptions:
            runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

运行AsUserName 字段可接受值的示例： ContainerAdministrator、、NT AUTHORITY\ N ContainerUser ETWORK SERVICE、NT AUTHORITY\ LOC

通常使用适用于 Windows pod 的容器来运行容 ContainerUser 器是个好主意。用户不在容器和主机之间共享，但在容器中 ContainerAdministrator 确实具有其他权限。请注意，有一些用户名限制需要注意。

何时使用的一个很好的例子 ContainerAdministrator 是设置 PATH。你可以使用 USER 指令来做到这一点，如下所示：

```
USER ContainerAdministrator
RUN setx /M PATH "%PATH%;C:/your/path"
USER ContainerUser
```

另请注意，机密以明文形式写在节点的卷上（与 linux 上的 tmpfs/in-memory 相比）。这意味着你必须做两件事

- 使用文件 ACL 保护机密文件的位置

- 使用卷级加密 BitLocker

# 永久存储选项

## 什么是树内和树外音量插件？

在引入容器存储接口 (CSI) 之前，所有卷插件都是树内的，这意味着它们是与核心 Kubernetes 二进制文件一起构建、链接、编译和发布的，并扩展了核心 Kubernetes API。这意味着向 Kubernetes（卷插件）添加新的存储系统需要将代码签入核心 Kubernetes 代码存储库。

Out-of-tree 卷插件是独立于 Kubernetes 代码库开发的，并作为扩展部署（安装）在 Kubernetes 集群上。这使供应商能够在带外更新驱动程序，即与 Kubernetes 发布周期分开更新。这在很大程度上是可能的，因为 Kubernetes 创建了一个存储接口或 CSI，为供应商提供了一种与 k8s 接口的标准方式。

你可以在上查看有关 Amazon Elastic Kubernetes 服务 (EKS) 存储类和 CSI 驱动程序的更多信息 https://docs.aws.amazon.com/eks/latest/userguide/storage.html

## In-tree 适用于 Windows 的音量插件

Kubernetes 卷允许在 Kubernetes 上部署具有数据持久性要求的应用程序。永久卷的管理包括 provisioning/de-provisioning/resizing 卷、一个卷、 attaching/detaching to/from 一个 Kubernetes 节点和 pod 中的 mounting/dismounting to/from 单个卷容器。用于为特定存储后端或协议实现这些卷管理操作的代码以 Kubernetes 卷插件（卷插件）In-tree 的形式提供。在亚马逊 Elastic Kubernetes 服务 (EKS) 上，Windows 支持以下类别的 Kubernetes 卷插件：

In-tree 音量插件：aw s ElasticBlockStore

为了在 Windows 节点上使用 In-tree 音量插件，需要创建一个额外的插件 StorageClass 来使用 NTFS 作为 fStype。在 EKS 上，默认值 StorageClass 使用 ext4 作为默认 fStype。

A StorageClass 为管理员提供了一种描述他们提供的存储 “类别” 的方法。不同的类别可能会映射到服务质量级别、备份策略或群集管理员确定的任意策略。Kubernetes 对类代表什么并不固执己见。在其他存储系统中，此概念有时被称为 “配置文件”。

你可以通过运行以下命令来检查它：

```
kubectl describe storageclass gp2
```

输出：

```
Name:            gp2
IsDefaultClass:  Yes
Annotations:     kubectl.kubernetes.io/last-applied-
configuration={"apiVersion":"storage.k8s.io/v1","kind":"StorageClas
","metadata":{"annotations":{"storageclass.kubernetes.io/is-default-
class":"true"},"name":"gp2"},"parameters":{"fsType"
"ext4","type":"gp2"},"provisioner":"kubernetes.io/aws-
ebs","volumeBindingMode":"WaitForFirstConsumer"}
,storageclass.kubernetes.io/is-default-class=true
Provisioner:           kubernetes.io/aws-ebs
Parameters:            fsType=ext4,type=gp2
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

要创建支持 NTFS 的新 StorageClass 版本，请使用以下清单：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gp2-windows
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ntfs
volumeBindingMode: WaitForFirstConsumer
```

StorageClass 通过运行以下命令创建：

```
kubectl apply -f NTFSStorageClass.yaml
```

下一步是创建永久卷声明 (PVC)。

PersistentVolume (PV) 是集群中由管理员配置或使用 PVC 动态配置的一块存储。它是群集中的资源，就像节点是群集资源一样。此 API 对象捕获存储实施的细节，无论是 NFS、iSCSI 还是特定于云提供商的存储系统。

A PersistentVolumeClaim (PVC) 是用户对存储的请求。索赔可以请求特定的大小和访问模式（例如，它们可以挂载 ReadWriteOnce， ReadOnlyMany 或 ReadWriteMany）。

对于不同的 PersistentVolumes 用例，用户需要不同的属性，例如性能。集群管理员需要能够提供各种不同的功能 PersistentVolumes ，而不仅仅是大小和访问模式，而不必让用户了解这些卷的实施细节。对于这些需求，有足够的 StorageClass 资源。

在下面的示例中，PVC 是在命名空间窗口中创建的。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-windows-pv-claim
  namespace: windows
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: gp2-windows
  resources:
    requests:
      storage: 1Gi
```

通过运行以下命令创建 PVC：

```
kubectl apply -f persistent-volume-claim.yaml
```

以下清单创建了一个 Windows Pod，将 as 设置 VolumeMount 为`C:\Data`并使用 PVC 作为附加存储`C:\Data`。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: windows-server-ltsc2019
  namespace: windows
spec:
  selector:
    matchLabels:
      app: windows-server-ltsc2019
      tier: backend
      track: stable
  replicas: 1
  template:
    metadata:
      labels:
        app: windows-server-ltsc2019
```

```
        tier: backend
        track: stable
    spec:
      containers:
      - name: windows-server-ltsc2019
        image: mcr.microsoft.com/windows/servercore:ltsc2019
        ports:
        - name: http
          containerPort: 80
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - mountPath: "C:\\data"
          name: test-volume
      volumes:
        - name: test-volume
          persistentVolumeClaim:
            claimName: ebs-windows-pv-claim
      nodeSelector:
        kubernetes.io/os: windows
        node.kubernetes.io/windows-build: '10.0.17763'
```

通过以下方式访问 Windows 窗格来测试结果 PowerShell：

```
kubectl exec -it podname powershell -n windows
```

在 Windows 窗格中，运行：`ls`

输出：

```
PS C:\> ls


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          3/8/2021   1:54 PM                data
d-----          3/8/2021   3:37 PM                inetpub
d-r---          1/9/2021   7:26 AM                Program Files
d-----          1/9/2021   7:18 AM                Program Files (x86)
d-r---          1/9/2021   7:28 AM                Users
d-----          3/8/2021   3:36 PM                var
```

```
d-----          3/8/2021   3:36 PM                Windows
-a----         12/7/2019   4:20 AM           5510 License.txt
```

数据目录由 EBS 卷提供。

## Out-of-tree 适用于 Windows

与 CSI 插件关联的代码以树外脚本和二进制文件形式发布，这些脚本和二进制文件通常作为容器镜像分发，并使用标准 Kubernetes 构造（如和）进行部署。 DaemonSets StatefulSetsCSI 插件可以处理 Kubernetes 中的各种卷管理操作。CSI 插件通常由节点插件（作为一个节点在每个节点上运行 DaemonSet）和控制器插件组成。

CSI 节点插件（尤其是那些与作为块设备或通过共享文件系统公开的永久卷相关的插件）需要执行各种特权操作，例如扫描磁盘设备、挂载文件系统等。这些操作因每个主机操作系统而异。对于 Linux 工作节点，容器化 CSI 节点插件通常部署为特权容器。对于 Windows 工作节点，使用 csi-proxy 支持容器化 CSI 节点插件的特权操作，csi-proxy 是一个由社区管理的独立二进制文件，需要预先安装在每个 Windows 节点上。

亚马逊 EKS 优化的 Windows AMI 包括从 2022 年 4 月 CSI-proxy 开始。客户可以使用 Windows 节点上的 SMB CSI 驱动程序访问适用于 Windows 文件服务器的 Amazon FSx、适用于 ONTAP 中小型企业共享的 Amazon FSx、 AWS and/or Storage Gateway — 文件网关。 NetApp

以下博客详细介绍了如何设置 SMB CSI 驱动程序以使用适用于 Windows File Server 的 Amazon FSx 作为 Windows Pod 的永久存储。

## 适用于 Windows File Server 的 Amazon FSx

一种选择是通过名为 SMB 全局映射的 SMB 功能使用适用于 Windows File Server 的 Amazon FS x，该功能允许在主机上挂载 SMB 共享，然后将该共享上的目录传递到容器中。容器不需要配置特定的服务器、共享、用户名或密码，而这些都是在主机上处理的。该容器的工作原理与本地存储相同。

> SMB Global Mapping 对协调器是透明的，并且通过它进行安装， HostPath 这可能意味着存在安全问题。

在下面的示例中，路径`G:\Directory\app-state`是 Windows 节点上的 SMB 共享。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-fsx
```

```
spec:
  containers:
  - name: test-fsx
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command:
      - powershell.exe
      - -command
      - "Add-WindowsFeature Web-Server; Invoke-WebRequest -UseBasicParsing
 -Uri 'https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.6/
ServiceMonitor.exe' -OutFile 'C:\\ServiceMonitor.exe'; echo '<html><body><br/
><br/><marquee><H1>Hello EKS!!!<H1><marquee></body><html>' > C:\\inetpub\\wwwroot\
\default.html; C:\\ServiceMonitor.exe 'w3svc'; "
    volumeMounts:
      - mountPath: C:\dotnetapp\app-state
        name: test-mount
  volumes:
    - name: test-mount
      hostPath:
        path: G:\Directory\app-state
        type: Directory
  nodeSelector:
      beta.kubernetes.io/os: windows
      beta.kubernetes.io/arch: amd64
```

以下博客详细介绍了如何将适用于 Windows File Server 的 Amazon FSx 设置为 Windows Pod 的永久存储空间。

# 强化 Windows 容器镜像

你是否在强化你的 Windows 容器镜像？ 多年来，我与全球客户合作，帮助他们将传统工作负载迁移到容器，尤其是 Windows 工作负载。凭借20多年的经验，我看到各组织投入了大量精力和资源来强化其Windows服务器，实施了从CIS基准测试到运行时防病毒保护的所有内容，以保护敏感数据。

然而，出现了一个令人担忧的趋势。随着这些高度安全的虚拟机被现代化为容器，许多关键的强化实践都被忽视了。Windows 安全最佳实践，从基础映像 (OS) 到 Web 服务（例如 IIS），经常被忽视，大部分重点完全放在保护容器主机上。必须认识到，虽然容器在隔离的命名空间中运行，但它们仍然与主机共享内核原语。攻击者通常对横向移动更感兴趣，而不是直接瞄准容器主机，这使他们能够利用薄弱的容器安全设置并访问敏感数据。

这些文档的目标是重点介绍您应该专门为在 IIS 上托管ASP.NET网站的 Windows 容器实现的一些基本安全设置。我们将重点关注四个关键领域：

- 账户安全政策
- 审计策略
- IIS 安全最佳实践
- 最低特权原则

首先，我们将深入探讨为什么这些安全配置对于保护您的 Windows 容器至关重要，并研究它们可以缓解的具体风险以及它们提供的安全优势。接下来，我们将演示如何在 Dockerfile 中正确实现这些配置的代码片段，确保您的容器能够抵御潜在威胁。最后，我们将详细分解每个设置，全面解释其功能、对容器安全的影响以及它如何有助于保护您的应用程序。这种方法不仅会向您展示如何应用这些最佳实践，还可以让您深入了解为什么它们对于在容器化环境中保持稳健的安全态势至关重要。

## 1. 使用本地安全策略和注册表配置账户策略（密码或锁定）

Windows Server Core 是一个最低安装选项，可作为 [EKS 优化的 Windows AMI] (https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-windows-ami.html) 的一部分提供。使用本地安全策略和注册表配置帐户策略（密码或锁定）通过强制执行可靠的密码和锁定规则来增强系统的安全性。这些策略要求用户创建具有明确规定的最小长度和复杂性的强密码，以防范与密码相关的常见攻击。

通过设置密码的最大使用年限，系统会提示用户定期更新密码，从而降低凭据泄露的可能性。锁定策略通过在指定次数的登录尝试失败后临时锁定帐户，从而增加了一层额外的保护，有助于防止暴力攻击。通过 Windows Registry 配置这些设置允许管理员在系统级别强制执行这些安全措施，从而确保整个组织的一致性和合规性。在 Windows 容器中应用这些账户策略对于保持安全一致性至关重要，尽管容器通常是临时性的，并且专用于隔离的工作负载：

### 安全一致性

- 合规性：在容器中实施一致的密码策略和锁定规则有助于保持安全合规性，尤其是在需要严格访问控制的环境中（例如 HIPAA 等合规性）。 PCI-DSS
- 强化容器：应用这些设置可确保您的 Windows 容器免受未经授权的访问或基于密码的攻击，从而使容器的安全状况与更广泛的系统安全策略保持一致。

### 防范暴力攻击

- 账户锁定：这些设置通过在特定次数的登录尝试失败后锁定账户，帮助抵御暴力登录尝试。这样可以防止攻击者尝试无限数量的密码。

- 密码复杂性：要求具有足够长度的复杂密码可以降低弱密码被利用的可能性，即使在孤立的容器化环境中也是如此。

## Multi-User 场景

- 如果您的容器化应用程序旨在处理多个用户或需要用户身份验证，则强制执行密码策略可确保容器内的用户帐户遵守严格的安全规则，从而限制只有经过授权的用户才能进行访问。

## 永久性 Windows 容器

- 虽然容器通常被认为是临时性的，但某些 Windows 容器可以运行长期服务或处理用户管理，因此强制执行与常规 Windows 服务器类似的适当安全策略非常重要。

## 混合环境中的一致性

- 如果您在基础架构中同时运行虚拟机和容器，则在所有环境中应用相同的安全策略（例如 password/lockout 策略）可确保统一的安全标准，从而简化监管和管理。

总而言之，在 Windows 容器中应用这些账户策略可确保您的容器不会成为安全策略中的弱点，从而防止密码攻击，并在整个环境中强制保持一致性。

Dockerfile：

```
# Configure account policies for password complexity and lockout
RUN powershell -Command \
      "Write-Output 'Configuring Account Policies (Password/Lockout)...'; \
      NET ACCOUNTS /MINPWLEN:14 /MAXPWAGE:60 /MINPWAGE:14 /LOCKOUTTHRESHOLD:5
```

解释：

本节通过 Windows 注册表配置密码和锁定设置的帐户策略。这些策略通过控制密码要求和账户锁定阈值来帮助加强安全性。

1. MinimumPasswordLength (MINPWLEN) = 14 此设置定义了密码的最小字符数。范围为 0-14 个字符；默认为六个字符。
2. MaximumPasswordAge (MAXPWAGE) = 60 此设置定义了密码有效的最大天数。使用 UNLIMITED 不会指定任何限制。/MAXPWAGE 不能小于 /MINPWAGE。范围为 1-999；默认值为 90 天

3. 锁定阈值（LOCKOUTTTHRESHOUT）= 5 此设置定义了登录尝试失败的阈值。在 5 次错误尝试后，该帐户将被锁定。

这些设置通过强制执行严格的密码策略和在一定次数的登录尝试失败后锁定帐户，有助于提高密码安全性并防止暴力攻击。

## 2. 审计策略

审计策略对于 Windows Containers 很重要，因为它们为登录尝试和权限使用等安全事件提供了重要的可见性，有助于检测未经授权的访问、监控用户活动并确保遵守监管标准。即使容器的短暂性质，审计日志对于事件调查、主动威胁检测以及在容器化环境中保持一致的安全态势也至关重要。

安全监控和合规性：

- 跟踪用户活动：审计策略允许管理员监控容器内的用户活动，例如登录尝试和权限使用。这对于检测未经授权的访问或可疑行为至关重要。
- 监管合规：许多组织都需要记录安全事件，以遵守 HIPAA 和 GDPR PCI-DSS 等法规。在容器中启用审计策略可确保您满足这些要求，即使在容器化环境中也是如此。

事件调查：

- 取证和分析：如果容器化应用程序或服务遭到入侵，审计日志可以为事后分析提供宝贵的见解。它们可以帮助安全团队追踪攻击者采取的行动或识别漏洞是如何发生的。
- Real-time 检测：审计日志允许管理员为关键事件（例如登录尝试失败、权限升级）设置实时警报。这种主动监控有助于及早发现攻击，并缩短响应时间。

跨环境的一致性：

- 统一的安全态势：通过注册表在容器中应用审计策略，可以确保容器化和非容器化环境中的安全实践保持一致。这样可以避免容器成为安全监控的盲点。
- 混合环境中的可见性：对于同时运行传统 Windows 服务器和容器的组织，审计策略可在所有平台上提供相似的可见性和控制力，从而使管理更轻松、更有效。

跟踪特权操作：

- 权限使用审计：在应用程序以更高的权限运行或执行管理任务的容器环境中，审计特权操作可确保问责制。您可以记录谁访问了容器内的敏感资源或执行了关键任务。

- 防止滥用权限：通过监控权限使用情况，您可以检测未经授权的用户何时尝试提升其权限或访问容器内的受限区域，这有助于防止内部或外部攻击。

检测未经授权的访问尝试：

- 登录尝试失败：为失败的登录尝试启用审计策略有助于识别暴力攻击或未经授权的访问容器化应用程序的尝试。这样可以了解谁在尝试访问系统以及访问频率。
- 账户锁定监控：通过审计账户锁定事件，管理员可以检测和调查可疑或恶意活动导致的潜在封锁。

即使在短暂的环境中也能保持持久的安全性：

- 短暂但安全：虽然容器是短暂的，这意味着它们可以经常被删除和重新创建，但审计在确保在容器运行时捕获安全事件方面仍然起着关键作用。这样可以确保在容器的生命周期内记录关键安全事件。

集中式日志：

- 将日志转发到集中式系统：容器可以与集中式日志系统（例如 ELK stack、AWS CloudWatch）集成，以捕获来自多个容器实例的审计日志。这样可以更好地分析和关联整个基础架构中的安全事件。

Dockerfile：

```
# Configure audit policies for logging security events
RUN powershell -Command \
    "Write-Host 'Configuring Audit Policy..'; \
    Set-ItemProperty -Path 'HKLM:\\SYSTEM\\CurrentControlSet\\Control\\Lsa' -Name
 'SCENoApplyLegacyAuditPolicy' -Value 0; \
    auditpol /set /category:"Logon/Logoff" /subcategory:"Logon" /failure:enable

# Creates STDOUT on Windows Containers (check GitHub LogMonitor:: https://github.com/
microsoft/windows-container-tools/blob/main/LogMonitor/README.md)
COPY LogMonitor.exe LogMonitorConfig.json 'C:\\LogMonitor\\'
WORKDIR /LogMonitor
```

解释：

本节使用注册表修改来配置审计策略。审计策略控制 Windows 记录哪些安全事件，这有助于监控和检测未经授权的访问尝试。

1. SCENoApplyLegacyAuditPolicy = 0 这将禁用旧版审计策略格式，从而启用更高版本的 Windows 中引入的更精细的审计策略。这对于现代审计配置非常重要。
2. Auditpol 子类别：“登录” 此设置允许对成功和失败登录事件进行审计。值 3 表示 Windows 将记录成功和失败的登录尝试。这有助于监控谁在访问系统并捕捉失败的登录尝试。

这些审计策略对于安全监控和合规性至关重要，因为它们提供了重要安全事件的详细日志，例如登录尝试和使用特权操作。

## 3. Windows 容器的 IIS 安全最佳实践

在 Windows 容器中实施 IIS 最佳实践很重要，原因有很多，可以确保您的应用程序安全、高性能和可扩展。尽管容器提供了隔离和轻量级环境，但它们仍然需要适当的配置以避免漏洞和操作问题。以下是为什么在 Windows 容器中遵循 IIS 最佳实践至关重要的原因：

安全性

- 预防常见漏洞：IIS 通常是跨站脚本 (XSS)、点击劫持和信息泄露等攻击的目标。实施安全标头（例如 X-Content-Type-Options X-Frame-Options、和 Strict-Transport-Security）有助于保护您的应用程序免受这些威胁的侵害。
- 隔离还不够：容器是隔离的，但是配置错误的 IIS 实例可能会暴露敏感信息，例如服务器版本详细信息、目录列表或未加密的通信。通过禁用目录浏览和删除 IIS 版本标头等功能，可以最大限度地减少攻击面。
- 加密和 HTTPS：最佳实践（例如强制 HTTPS-only 连接）可确保传输中的数据经过加密，从而保护敏感信息免遭拦截。

性能

- 高效使用资源：IIS 最佳实践，例如启用动态和静态压缩，可减少带宽使用量并缩短加载时间。这些优化在容器化环境中尤为重要，在这种环境中，资源在容器和主机系统之间共享。
- 优化的日志记录：正确配置日志记录（例如，包括 X-Forwarded-For 标头）可确保您可以跟踪客户端活动，同时最大限度地减少不必要的日志记录开销。这可以帮助您收集相关数据以进行故障排除，而不会降低性能。

可扩展性和可维护性

- 跨环境的一致性：通过遵循最佳实践，可以确保 IIS 配置在多个容器实例之间保持一致。这简化了扩展，并确保在部署新容器时，它们遵守相同的安全和性能准则。

- 自动配置：Dockerfiles 中的最佳实践，例如设置文件夹权限和禁用不必要的功能，可确保每个新容器都自动正确配置。这减少了手动干预，降低了人为错误的风险。

合规

- 满足监管要求：许多行业都有严格的监管要求（例如 HIPAA） PCI-DSS，要求采取特定的安全措施，例如加密通信 (HTTPS) 和记录客户请求。在容器中遵循 IIS 最佳实践有助于确保遵守这些标准。
- 可审计性：实施审计策略和安全日志可以实现事件的可追溯性，这在审计中至关重要。例如，记录 X-Forwarded-For 标头可确保在基于代理的架构中正确记录客户端 IP 地址。

最大限度地降低共享环境中的风险

- 避免配置错误：容器共享主机的内核，虽然它们彼此隔离，但配置不当的 IIS 实例可能会暴露漏洞或造成性能瓶颈。最佳实践可确保每个 IIS 实例以最佳方式运行，从而降低跨容器问题的风险。
- 最低权限访问权限：为容器内的文件夹和文件设置适当的权限（例如， Set-Acl 在中使用 PowerShell）可确保容器内的用户和进程只有必要的访问权限，从而降低权限升级或数据被篡改的风险。

短暂环境中的弹性

- 容器的短暂性质：容器通常是短暂的，并且经常重建。应用 IIS 最佳实践可确保每个容器无论重新部署多少次，都能安全一致地进行配置。这样可以防止随着时间的推移而引入错误配置。
- 缓解潜在的错误配置：通过自动实施最佳实践（例如，禁用弱协议或标头），可以最大限度地降低容器重启或更新期间出现错误配置的风险。

Dockerfile：

```
# Enforce HTTPS (disable HTTP) -- Only if container is target for SSL termination
RUN powershell -Command \
    "$httpBinding = Get-WebBinding -Name 'Default Web Site' -Protocol http | Where-
Object { $_.bindingInformation -eq '*:80:' }; \
    if ($httpBinding) { Remove-WebBinding -Name 'Default Web Site' -Protocol http -Port
 80; } \
    $httpsBinding = Get-WebBinding -Name 'Default Web Site' -Protocol https | Where-
Object { $_.bindingInformation -eq '*:443:' }; \
```

```
    if (-not $httpsBinding) { New-WebBinding -Name 'Default Web Site' -Protocol https -
Port 443 -IPAddress '*'; }"

# Use secure headers
RUN powershell -Command \
    "Write-Host 'Adding security headers...'; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 'system.applicationHost/sites/siteDefaults/logFile/customFields' -name
 "." -value @{logFieldName='X-Forwarded-For';sourceName='X-Forwarded-
For';sourceType='RequestHeader'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='Strict-
Transport-Security';value='max-age=31536000; includeSubDomains'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Content-Type-
Options';value='nosniff'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-XSS-
Protection';value='1; mode=block'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Frame-
Options';value='DENY'};"

# Disable IIS version disclosure
RUN powershell -Command \
    "Write-Host 'Disabling IIS version disclosure...'; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/security/requestFiltering" -name "removeServerHeader" -value
 "true";"

# Set IIS Logging Best Practices
RUN powershell -Command \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/directoryBrowse" -name "enabled" -value "false"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpErrors" -name "existingResponse" -value "PassThrough"; \

# Enable IIS dynamic and static compression to optimize performance
RUN powershell -Command \
    "Write-Host 'Enabling IIS compression...'; \
    Enable-WindowsOptionalFeature -Online -FeatureName IIS-HttpCompressionDynamic; \
    Import-Module WebAdministration; \
```

```
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doDynamicCompression" -value "true"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doStaticCompression" -value "true"

# Ensure proper folder permissions using PowerShell's Set-Acl

RUN powershell -Command \
    "Write-Host 'Setting folder permissions for IIS...'; \
    $path = 'C:\\inetpub\\wwwroot'; \
    $acl = Get-Acl $path; \
    $iusr = New-Object System.Security.Principal.NTAccount('IIS_IUSRS'); \
    $rule = New-Object System.Security.AccessControl.FileSystemAccessRule($iusr,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule); \
    $users = New-Object System.Security.Principal.NTAccount('Users'); \
    $rule2 = New-Object System.Security.AccessControl.FileSystemAccessRule($users,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule2); \
    Set-Acl -Path $path -AclObject $acl"
```

解释：

此命令将 IIS 配置为记录 X-Forwarded-For 标头，该标头通常用于在请求通过代理或负载均衡器时捕获原始客户端 IP 地址。默认情况下，IIS 仅记录负载均衡器或反向代理的 IP 地址，因此添加此自定义日志字段有助于跟踪真实的客户端 IP，以便进行安全审计、分析和故障排除。

1. X-Forwarded-For 标头，通常用于在请求通过代理或负载均衡器时捕获原始客户端 IP 地址。默认情况下，IIS 仅记录负载均衡器或反向代理的 IP 地址，因此添加此自定义日志字段有助于跟踪真实的客户端 IP，以便进行安全审计、分析和故障排除。
2. Strict-Transport-Security (HSTS) 确保浏览器仅通过 HTTPS 进行通信。max-age=31536000 规定此政策的有效期为 1 年，并且包含SubDomains 将该政策应用于所有子域名。
3. X-Content-Type-Options防止浏览器 “MIME-sniffing” 的响应远离声明的 Content-Type。这有助于防止某些类型的攻击。
4. X-XSS-Protection在浏览器中启用 Cross-Site 脚本 (XSS) 保护。
5. X-Frame-Options防止页面嵌入到 iframe 中，防止点击劫持攻击。
6. 禁用 IIS 版本泄露此命令禁用 HTTP 响应中的服务器标头，默认情况下，服务器标头会显示正在使用的 IIS 版本。隐藏这些信息有助于降低攻击者识别和瞄准 IIS 版本特有的漏洞的风险。

7. 启用 HTTPS-only 连接此（已注释）部分强制执行 HTTPS 连接并禁用 HTTP。如果取消注释，Dockerfile 会将 IIS 配置为仅在端口 443 (HTTPS) 上侦听，并删除端口 80 上的默认 HTTP 绑定。这在终止容器内的 SSL 时很有用，可以确保所有流量都经过加密。
8. 禁用目录浏览可防止 IIS 在没有默认文档的情况下显示目录列表。这样可以避免向用户公开内部文件结构。
9. 传递自定义错误页面确保如果应用程序有自己的错误处理功能，IIS 将允许应用程序的错误页面通过，而不是显示默认的 IIS 错误页面。
10. 详细错误模式将 IIS 配置为仅显示本地请求的详细错误消息，从而帮助开发人员在不向外部用户泄露敏感信息的情况下诊断问题。
11. 确保正确的文件夹权限此区块为 IIS Web 根目录 (C:\inetpub\wwwroot) 配置文件夹权限。它为 IIS_IUSRS 和 Users 组设置 “读取” 和 “执行” 权限，确保这些用户可以访问该文件夹，但不能修改文件。设置正确的权限可以最大限度地降低未经授权访问或篡改 Web 服务器托管文件的风险。

在 Windows 容器中遵循 IIS 最佳实践可确保您的容器化应用程序安全、高性能和可扩展。这些做法有助于防止漏洞、优化资源使用、确保合规性并保持容器实例之间的一致性。尽管容器设计为隔离式，但必须进行适当的配置，以最大限度地降低风险并确保应用程序在生产环境中的可靠性。

## 4. 最低特权原则

最低权限原则 (PoLP) 对于 Windows 容器至关重要，原因有很多，特别是在容器化环境中增强安全性和最大限度地降低风险方面。该原则规定，系统或应用程序应以正常运行所需的最低权限级别运行。以下是它在 Windows 容器中很重要的原因：

最大限度地减少攻击面

- 容器通常运行与各种系统组件交互的应用程序，应用程序拥有的权限越多，它对这些组件的访问权限就越广。通过将容器的权限限制在必要的范围内，PoLP 可以显著缩小攻击面，使攻击者在容器受到威胁时更难利用容器。

限制受损容器的影响

- 如果 Windows 容器遭到入侵，运行具有过高权限（例如管理员或根级访问权限）的应用程序可能会使攻击者获得对关键系统文件的控制权或在容器主机上升级权限。通过强制执行 PoLP，即使容器遭到入侵，攻击者也能做的事情受到限制，从而防止进一步升级和访问敏感资源或其他容器。

多租户环境中的保护

- 在云或企业环境中，多个容器可以在同一个物理或虚拟基础架构上运行。PoLP 可确保受感染的容器无法访问属于其他租户的资源或数据。这种隔离对于维护共享的多租户环境中的安全性以及防止容器之间的横向移动至关重要。

缓解权限升级

- 攻击者可以利用以高权限运行的容器来提升系统内的权限。PoLP 通过限制容器对系统资源的访问来降低这种风险，从而防止在容器环境之外进行未经授权的操作或权限升级。

合规与审计

- 许多监管标准和安全框架（例如 PCI DSS、HIPAA、GDPR）都要求系统遵守 PoLP，以限制对敏感数据的访问。以受限权限运行 Windows 容器可以帮助组织遵守这些法规，并确保应用程序只能访问他们特别需要的资源。

降低配置错误的风险

- 当容器以不必要的权限运行时，即使是轻微的配置错误也可能导致严重的安全漏洞。例如，如果以管理员身份运行的容器意外暴露在互联网上，则攻击者可以控制系统。PoLP 通过默认使用有限权限来帮助防止此类风险，从而降低错误配置的危险。

改善了容器安全状况

- 通过遵循 PoLP，可以更好地将容器与底层主机系统隔离开来，并且可以相互隔离。这可确保容器化应用程序不太可能访问或修改超出其定义范围的系统文件或进程，从而保持主机操作系统和其他工作负载的完整性。

Dockerfile：

```
# Strongly recommended that when deploying a Windows server container to any multi-
tenant environment that your application runs via the ContainerUser account
USER ContainerUser
```

解释：

在本节中，USE ContainerUser R 命令指定 Windows 容器内的应用程序应使用该 ContainerUser 帐户而不是默认管理员帐户运行。

以下是为什么这很重要，尤其是在多租户环境中：

1. 最低权限原则：该 ContainerUser 帐户是具有有限权限的非管理员用户。使用此帐户运行应用程序遵循最低权限原则，这有助于最大限度地降低被利用的风险。如果攻击者破坏应用程序，他们对系统的访问权限将受到限制，从而减少潜在的损害。
2. 增强安全性：在多租户环境中，容器可以共享相同的底层基础架构。以身份运行 ContainerUser 可确保即使一个容器遭到入侵，它也没有访问或修改关键系统文件或其他容器的管理权限。这会显著减少攻击面。
3. 避免 root 访问权限：默认情况下，容器可能以更高的权限运行（类似于 Linux 容器中的 root 访问权限），如果被利用，可能会很危险。使用 ContainerUser 可确保应用程序不会以不必要的管理权限运行，从而使攻击者更难升级权限。
4. 多租户环境的最佳实践：在多个用户或组织共享相同基础架构（例如云中）的环境中，安全性至关重要。以受限权限运行应用程序可以防止一个租户的应用程序影响其他租户，从而保护整个平台上的敏感数据和资源。

US E ContainerUser R 命令可确保应用程序以最低权限运行，通过限制容器受损可能造成的损害，增强多租户环境中的安全性。这是在容器化环境中防止未经授权的访问或权限升级的最佳实践。

最低权限原则对于 Windows 容器至关重要，因为它可以限制安全漏洞的潜在影响，减少攻击面，并防止未经授权访问关键系统组件。通过仅以必要权限运行容器化应用程序，组织可以显著增强其容器环境的安全性和稳定性，尤其是在多租户和共享基础架构中。

## 最后的想法：为什么保护你的 Windows 容器是当今威胁格局 Must-Have 中的一个问题

在当今快速发展的数字世界中，威胁变得越来越复杂和丰富，保护你的 Windows 容器不仅仅是一项建议，而且绝对是必要的。容器提供了一种轻量、灵活的方式来打包和部署应用程序，但它们也无法幸免于安全漏洞。随着越来越多的企业采用容器来简化其基础架构，如果安全措施不当，它们也会成为网络攻击的潜在目标。

互联网充斥着各种威胁，从针对未修补漏洞的恶意行为者到自动扫描错误配置的机器人。如果没有适当的安全措施，容器可能会被利用来泄露敏感数据、提升权限或成为可能危及更广泛基础架构的攻击的入口点。这使得容器安全与保护环境的任何其他部分一样重要。

使用 Windows 容器时，许多传统的安全最佳实践仍然适用。实施强大的账户策略、保护 IIS 配置、强制执行 HTTPS、使用严格的防火墙规则以及对关键文件应用最低权限访问权限都是确保容器能够抵御攻击的关键措施。此外，定期的审计和记录可以让你了解集装箱内正在发生的事情，从而使你能够在可疑活动演变为全面的事件之前将其捕获。

保护 Windows 容器还符合要求保护敏感数据和确保应用程序完整性的监管要求。随着云原生架构和容器化架构变得越来越普遍，确保从基础映像到正在运行的容器等每一层的安全性将有助于保护您的运营并保持客户信任。

总而言之，容器化应用程序的兴起，加上越来越多的网络威胁，使容器安全成为现代基础设施管理中不可谈判的方面。通过遵循最佳实践并持续监控漏洞，企业可以在不影响安全性的情况下享受 Windows 容器的敏捷性和效率。在这个威胁密集的环境中，保护你的 Windows 容器不仅仅是一种选择，更是必不可少的。

# 混合部署的最佳实践

本指南提供了有关使用 EKS 混合节点或 EKS Anywhere 在本地或边缘环境中运行部署的指导。

我们目前已经发布了以下主题的指南：

- EKS 混合节点和网络断开的最佳实践

# EKS 混合节点和网络断开连接

对于习惯于完全在自己的数据中心或边缘位置运行本地 Kubernetes 集群的客户来说，EKS 混合节点架构可能是全新的架构。使用 EKS 混合节点，Kubernetes 控制平面在 AWS 区域中运行，并且只有节点在本地运行，从而形成“延伸”或“扩展”的 Kubernetes 集群架构。

这就引出了一个常见的问题：“如果我的节点与 Kubernetes 控制平面断开连接会发生什么？”

在本指南中，我们通过回顾以下主题来回答这个问题。建议通过断开网络来验证应用程序的稳定性和可靠性，因为每个应用程序的行为可能因其依赖关系、配置和环境而有所不同。有关测试设置、程序和结果，请参阅 aws samples/eks--hybrid-exam GitHub ples 存储库，您可以参考测试使用 EKS 混合节点和您自己的应用程序的网络断开连接。该 GitHub 存储库还包含用于验证本指南中所述行为的测试的更多详细信息。

- 通过网络断开连接实现稳定性的最佳实践
- 通过网络断开连接实现的 Kubernetes 容器故障转移行为
- 通过网络断开连接产生的应用程序网络流量
- 通过网络断开连接获得的主机凭证

## 通过网络断开连接实现稳定性的最佳实践

### 高度可用的网络

避免混合节点和 Kubernetes 控制平面之间网络断开的最佳方法是使用本地环境与 AWS 之间的冗余弹性连接。有关使用这些解决方案构建高可用性混合网络的更多信息，请参阅 AWS Di rect Connect Connect 弹性工具包和 AWS Site-to-Site VPN 文档。

## 高可用性应用程序

在设计应用程序时，请考虑您的故障域以及不同类型的中断的影响。Kubernetes 提供了跨节点、区域和区域域部署和维护应用程序副本的内置机制。这些机制的使用取决于您的应用程序架构、环境和可用性要求。例如，无状态应用程序通常可以部署多个副本，并且可以在任意主机和基础架构容量之间移动，并且您可以使用节点选择器和拓扑分布限制在不同的域中运行应用程序的实例。有关在 Kubernetes 上构建弹性应用程序的应用程序级技术的详细信息，请参阅 EKS 最佳实践指南。

在确定是否将 Pod 移至其他节点时，Kubernetes 会评估与 Kubernetes 控制平面断开连接的节点的区域信息。如果一个区域中的所有节点都无法访问，Kubernetes 将取消该区域中节点的容器驱逐。作为最佳实践，如果您的部署中有节点在多个数据中心或物理位置运行，请根据每个节点的数据中心或物理位置为其分配一个区域。当您在云中的节点上运行 EKS 时，AWS 云控制器管理器会自动应用此区域标签。但是，混合节点不使用云控制器管理器，因此您可以通过 kubelet 配置传递这些信息。下面显示了如何在混合节点的节点配置中配置区域的示例。当您使用混合节点 CLI (`nodeadm`) 将混合节点连接到集群时，配置即被传递。有关`topology.kubernetes.io/zone`标签的更多信息，请参阅 Kubernetes 文档。有关混合节点 CLI 的更多信息，请参阅混合节点 nodeadm 参考。

```
apiVersion: node.eks.aws/v1alpha1
kind: NodeConfig
spec:
  cluster:
    name: my-cluster
    region: my-region
  kubelet:
    flags:
       - --node-labels=topology.kubernetes.io/zone=dc1
  hybrid:
    ...
```

## 网络监测

如果您使用 AWS Direct Connect 或 AWS Site-to-Site VPN 进行混合连接，则可以利用 CloudWatch 警报、日志和指标来观察混合连接的状态并诊断问题。有关更多信息，请参阅监控 AWS Direct Connect 资源和监控 AWS Site-to-Site VPN 连接。

建议为在 EKS 控制平面上运行的节点生命周期控制器报告`NodeNotReady`的事件创建警报，这些事件表示混合节点可能出现网络断开的情况。您可以通过以下方式创建此警报：为控制器管理器启用 EKS 控制平面日志记录，然后在 CloudWatch status= “” 中为 “记录节点的状态更改事件消息” 消息创建指标过滤器。`NodeNotReady`创建指标筛选器后，您可以根据所需的阈值为此过滤器创建警报。有关更多信息，请参阅 CloudWatch 文档中的日志警报。

您可以使用 Transit Gateway (TGW) 和虚拟专用网关 (VGW) 的内置指标来观察进出您的 TGW 或 VGW 的网络流量。您可以为这些指标创建警报，以检测网络流量降至正常水平以下的情况，这表明混合节点和 EKS 控制平面之间存在潜在的网络问题。下表描述了 TGW 和 VGW 指标。

| 网关 | 指标 | 说明 |
| --- | --- | --- |
| Transit Gateway | BytesIn | TGW 从附件（EKS 控制平面到混合节点）接收的字节 |
| Transit Gateway | BytesOut | 从 TGW 发送到附件（混合节点到 EKS 控制平面）的字节 |
| 虚拟专用网关 | TunnelDataIn | 从 AWS 端通过 VPN 隧道发送到客户网关（EKS 控制平面到混合节点）的字节 |
| 虚拟专用网关 | TunnelDataOut | 通过 VPN 隧道从客户网关（到 EKS 控制平面的混合节点）在连接的 AWS 端接收的字节 |

您还可以使用 CloudWatch Network Monit or 更深入地了解您的混合连接，从而缩短平均恢复时间，并确定网络问题是源于 AWS 还是您的环境。 CloudWatch Network Monitor 可用于可视化混合网络连接中的数据包丢失和延迟，设置警报和阈值，然后采取措施提高网络性能。有关更多信息，请参阅使用 Amazon CloudWatch 网络监视器。

EKS 提供了多种用于监控集群和应用程序运行状况的选项。对于集群运行状况，您可以使用 EKS 控制台中的可观察性控制面板来快速检测、故障排除和修复问题。您还可以使用适用于 Prometheus 的亚马逊托管服务、用于开放遥测 (ADOT) 的 AWS Distro 以及集群 CloudWatch 、应用程序和基础设施监控。有关 EKS 可观察性选项的更多信息，请参阅监控集群性能和查看日志。

## 本地故障排除

要为混合节点和 EKS 控制平面之间的网络断开做好准备，您可以设置辅助监控和日志后端，以便在无法访问区域 AWS 服务时保持应用程序的可观察性。例如，您可以将 AWS Distro 开放遥测版 (ADOT) 收集器配置为向多个后端发送指标和日志。您还可以使用本地工具（例如 `crictl` CLI）与 Pod 和容器进行本地交互，以替代通常查询 Kubernetes API 服务器端点的其他 Kubernetes API-compatible 客户端。kubectl有关的更多信息crictl，请参阅 cri- GitHub t crictlools 中的文档。下面列出了一些有用的crictl命令。

列出主机上运行的 pod：

```
crictl pods
```

列出主机上运行的容器：

```
crictl ps
```

列出主机上运行的映像：

```
crictl images
```

获取主机上运行的容器的日志：

```
crictl logs CONTAINER_NAME
```

获取主机上运行的 Pod 的统计信息：

```
crictl statsp
```

## 应用程序网络流量

使用混合节点时，重要的是要考虑和了解应用程序流量的网络流量，以及用于向集群外部公开应用程序的技术。在网络断开连接期间，不同的应用程序负载平衡和入口技术的行为会有所不同。例如，如果您使用 Cilium 的 BGP 控制平面功能进行应用程序负载平衡，则在网络断开连接期间，Pod 和服务的 BGP 会话可能会关闭。之所以发生这种情况，是因为 BGP 扬声器功能与 Cilium 代理集成在一起，而且 Cilium 代理在与 Kubernetes 控制平面断开连接时将持续重启。重启的原因是Cilium的运行状况检查失败，因为它的生命值与对Kubernetes控制平面的访问权限相结合（参见 CFP: #31702，C ilium v1.17 中有选择加入的改进）。同样，如果您使用应用程序负载均衡器 (ALB) 或网络负载均衡器 (NLB) 来处理 AWS Region-originated 应用程序流量，则如果您的本地环境与 AWS 区域断开连接，则该流量可能会暂时中断。建议在部署到生产环境之前，验证用于负载平衡和入口的技术在网络断开连接期间是否保持稳定。aws-h samples/eksybrid-exam GitHub ples 存储库中的示例使用 MetalLB 在 L2 模式下进行负载平衡，在混合节点和 EKS 控制平面之间的网络断开连接期间，负载平衡保持稳定。

## 查看对远程 AWS 服务的依赖关系

使用混合节点时，请注意您对本地或边缘环境外部的区域 AWS 服务的依赖性。示例包括访问 Amazon S3 或 Amazon RDS 获取应用程序数据、使用适用于 Prometheus 的亚马逊托管服务 CloudWatch 或指标和日志、使用应用程序和网络负载均衡 Region-originated 器处理流量，以及从 Amazon Elastic Container Registry 中提取容器。在您的本地环境与 AWS 之间断开网络连接期间，将无法访问这些服务。如果您的本地环境容易与 AWS 断开网络连接，请检查您对 AWS 服务的使用情况，并确保与这些服务的连接中断不会影响应用程序的静态稳定性。

## 调整 Kubernetes 容器组（pod）失效转移行为

对于无法跨主机移植的应用程序，或者没有备用容量进行容器故障转移的资源受限的环境，可以选择在网络断开连接期间调整容器故障转移行为。通常，重要的是要考虑应用程序的资源需求，并有足够的容量让一个或多个应用程序实例在节点出现故障时故障转移到另一台主机。

- 选项 1-使用 DaemonSets：此选项适用于可以而且应该在集群中的所有节点上运行的应用程序。DaemonSets 会自动配置为容忍无法访问的污点，这会通过网络断开连接使 DaemonSet Pod 与其节点绑定。
- 选项 2-调整无法访问`tolerationSeconds`的污点：您可以调整网络断开连接期间您的 Pod 与节点绑定的时间。为此，请将应用程序 pod 配置为在您指定的持续时间（在应用程序`NoExecute`规范`tolerationSeconds`中）内容忍无法达到的污点。使用此选项，当网络断开连接时，您的应用程序 Pod 将一直绑定到节点，直到`tolerationSeconds`到期。请仔细考虑这一点，因为增加无法访问`tolerationSeconds`的污点`NoExecute`意味着在无法访问的主机上运行的 Pod 可能需要更长的时间才能移动到其他可访问、运行良好的主机。
- 选项 3：自定义控制器：您可以创建并运行一个自定义控制器（或其他软件），用于监控 Kubernetes 中是否存在带有该效果的无法触及的污点。`NoExecute`当检测到这种污点时，自定义控制器可以检查应用程序特定的指标以评估应用程序的运行状况。如果应用程序运行正常，则自定义控制器可以移除无法访问的污点，从而防止在网络断开连接期间将 Pod 从节点中驱逐。

下面显示了如何`tolerationSeconds`为无法到达的污点配置部署的示例。在示例中，设置`tolerationSeconds`为`1800`（30 分钟），这意味着只有在网络断开连接持续时间超过 30 分钟时，在无法访问的节点上运行的 Pod 才会被驱逐出去。

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
      tolerations:
      - key: "node.kubernetes.io/unreachable"
        operator: "Exists"
        effect: "NoExecute"
        tolerationSeconds: 1800
```

# 通过网络断开连接实现的 Kubernetes 容器故障转移

我们首先回顾一下影响节点与 Kubernetes 控制平面之间网络断开连接期间 Kubernetes 行为的关键概念、组件和设置。EKS 符合上游 Kubernetes 标准，因此此处描述的所有 Kubernetes 概念、组件和设置都适用于 EKS 和 EKS 混合节点部署。

对 EKS 进行了一些改进，专门用于改善网络断开连接期间的 pod 故障转移行为，有关更多信息，请参阅上游 Kubernetes 存储库中的 GitHub 问题 #131294 和 #131481。

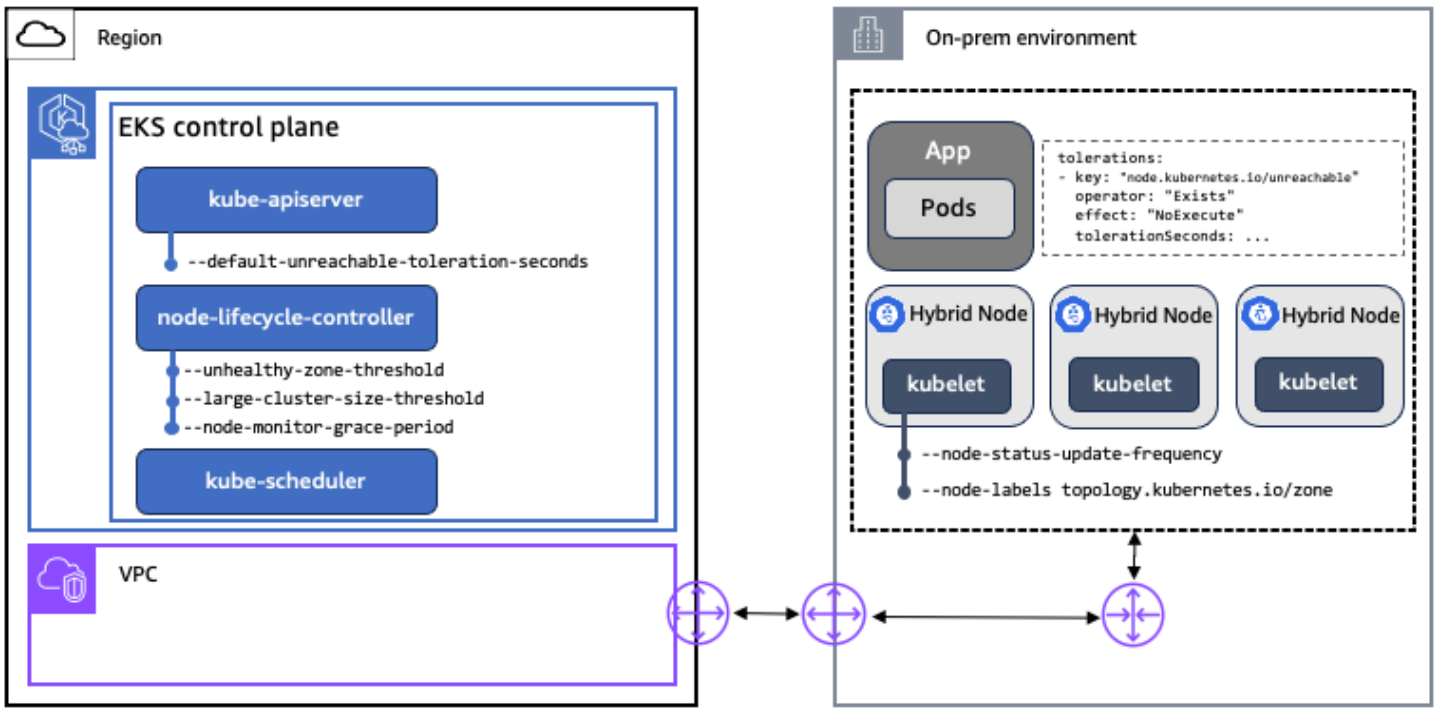
## 概念

污点和容忍度：Kubernetes 中使用污点和容忍度来控制 Pod 在节点上的调度。节点生命周期控制器设置污点，表示节点不符合调度条件，或者应该驱逐这些节点上的 Pod。当由于网络断开连接而无法访问节点时，节点生命周期控制器会应用 node.kubernetes。 io/unreachable 带有 NoSchedule 效果的污点，如果满足某些条件，则会 NoExecute 产生效果。node.kubernetes。 io/unreachable 污点对应于 NodeCondition Ready being Unknown。用户可以在中指定应用程序级别对污点的容忍度。 PodSpec

- NoSchedule: 除非有匹配的容忍度，否则不会在受污染的节点上调度新 Pod。已在节点上运行的 Pod 不会被驱逐出去。
- NoExecute: 不容忍污点的 Pod 会立即被驱逐。容忍污点（未指定 TolerationSeconds）的 Pod 将永远保持绑定。容忍具有指定 TolerationSeconds 污点的 Pod 将在指定的时间内保持绑定。在这段时间过后，节点生命周期控制器会将 Pod 逐出节点。

节点租约：Kubernetes 使用 Lease API 将 kubelet 节点心跳传送到 Kubernetes API 服务器。对于每个节点，都有一个名称匹配的 Lease 对象。在内部，每个 kubelet 心跳都会更新 Lease 对象的 spec.renewTime 字段。Kubernetes 控制平面使用该字段的时间戳来确定节点的可用性。如果节点与 Kubernetes 控制平面断开连接，则它们无法为其租约更新 spec.renewTime，控制平面会将其解释为 Ready being Unknown。 NodeCondition

## 组件




| 组件 | Sub-component | 说明 |
| --- | --- | --- |
| Kubernetes 控制平面 | kube-api 服务器 | API 服务器是 Kubernetes 控制平面的核心组件，用于公开 Kubernetes API。 |
| Kubernetes 控制平面 | 节点生命周期控制器 | kube控制器管理器运行的控制器之一。它负责检测和响应节点问题。 |
| Kubernetes 控制平面 | kube-scheduler | 一个控制平面组件，用于监视未分配节点的新创建的 Pod，并选择一个节点让它们运行。 |
| Kubernetes 节点 | kubelet | 在群集中的每个节点上运行的代理。kubelet 会监视 PodSpecs 并确保其中描述的容 PodSpecs 器运行良好。 |

## 配置设置

| 组件 | 设置 | 说明 | K8s 默认 | EKS 默认 | 可在 EKS 中配置 |
|---|---|---|---|---|---|
| kube-api 服务器 | 默认无法达到的容差秒数 | 表示tolerationSeconds 默认情况下，每个还没有unreachable:NoExecute 这种容忍度的 pod 都会添加该容忍度。 | 300 | 300 | 否 |
| 节点生命周期控制器 | 节点监视器宽限期 | 节点在被标记为不健康之前可能没有响应的时间。必须是 kubelet 的 N 倍nodeStatusUpdateFrequency ，其中 N 是 kubelet 发布节点状态时允许的重试次数。 | 40 | 40 | 否 |
| 节点生命周期控制器 | 大型集群大小阈值 | 节点生命周期控制器在驱逐逻辑中将集群视为大型集群的节点数量。 --secondary-node-eviction-rate 对于这种大小或更小的集群，则重写为 0。 | 50 | 100000 | 否 |
| 节点生命周期控制器 | 不健康区域阈值 | 区域中必须处于 “未就绪” 状态才能将该区域视为不健康的节点的百分比。 | 55% | 55% | 否 |
| kubelet | 节点状态更新频率 | kubelet 向控制平面发布节点状态的频率。必须与节点生命周期控制nodeMonitorGracePeriod 器兼容。 | 10 | 10 | 是 |
| kubelet | 节点标签 | 在集群中注册节点时要添加的标签。topology.kubernetes.io/zone 可以为混合节点指定标签，以将节点分组为多个区域。 | 无 | 无 | 是 |

## 通过网络断开连接实现的 Kubernetes 容器故障转移

此处描述的行为假设 Pod 以默认设置的 Kubernetes 部署形式运行，并且 EKS 被用作 Kubernetes 提供者。根据您的环境、网络断开连接的类型、应用程序、依赖关系和群集配置，实际行为可能会有所不同。本指南中的内容已使用特定的应用程序、集群配置和插件子集进行了验证。强烈建议在进入生产环境之前，在自己的环境和自己的应用程序中测试行为。

当节点和 Kubernetes 控制平面之间出现网络断开连接时，每个断开连接的节点上的 kubelet 将无法与 Kubernetes 控制平面通信。因此，在恢复连接之前，kubelet 无法驱逐这些节点上的 pod。这意味着，假设没有其他故障导致它们关闭，则在网络断开连接之前在这些节点上运行的 Pod 将在断开连接期间继续运行。总而言之，在节点与 Kubernetes 控制平面之间的网络断开连接期间，你可以实现静态稳定性，但是在恢复连接之前，你无法对节点或工作负载执行变更操作。

根据网络断开的性质，有五种主要场景会产生不同的 pod 故障转移行为。在所有情况下，一旦节点重新连接到 Kubernetes 控制平面，集群就会恢复健康，无需操作员干预。以下场景概述了基于我们的观察结果的预期结果，但这些结果可能不适用于所有可能的应用程序和集群配置。

场景 1：集群完全中断

预期结果：无法访问的节点上的 Pod 不会被驱逐并继续在这些节点上运行。

集群完全中断意味着集群中的所有节点都与 Kubernetes 控制平面断开连接。在这种情况下，控制平面上的节点生命周期控制器检测到集群中的所有节点都无法访问并取消任何 pod 驱逐。

在断开连接`Not Ready`期间，集群管理员将看到所有带有状态的节点。Pod 状态不会改变，在断开连接和后续重新连接期间，也不会在任何节点上调度新的 Pod。

场景 2：整个区域中断

预期结果：无法访问的节点上的 Pod 不会被驱逐并继续在这些节点上运行。

整个区域中断意味着该区域中的所有节点都与 Kubernetes 控制平面断开连接。在这种情况下，控制平面上的节点生命周期控制器检测到该区域中的所有节点都无法访问并取消任何 pod 驱逐。

在断开连接`Not Ready`期间，集群管理员将看到所有带有状态的节点。Pod 状态不会改变，在断开连接和后续重新连接期间，也不会在任何节点上调度新的 Pod。

场景 3：多数区域中断

预期结果：无法访问的节点上的 Pod 不会被驱逐并继续在这些节点上运行。

多数区域中断意味着给定区域中的大多数节点都与 Kubernetes 控制平面断开连接。Kubernetes 中的区域由具有相同标签的节点定义。`topology.kubernetes.io/zone`如果集群中未定义任何区域，则大多数中断意味着整个集群中的大多数节点都已断开连接。默认情况下，多数由节点生命周期控制器定义，在 Kubernetes 和 EK unhealthy-zone-threshold S 中均设置为 55%。由于在 EKS 中设置`large-cluster-size-threshold`为 100,000，因此如果一个区域中 55% 或更多的节点无法访问，则吊舱驱逐将被取消（假设大多数集群远小于 100,000 个节点）。

在断开连接`Not Ready`期间，集群管理员将看到区域中的大多数节点都处于状态状态，但是 Pod 的状态不会改变，也不会在其他节点上重新调度。

请注意，上述行为仅适用于大于三个节点的集群。在三个或更少节点的集群中，无法访问的节点上的 Pod 会被安排驱逐，而新 Pod 则被调度到健康的节点上。

在测试过程中，我们偶尔会观察到，在网络断开连接期间，Pod 会被逐出一个无法访问的节点，即使该区域的大多数节点都无法访问。我们仍在调查 Kubernetes 节点生命周期控制器中可能存在的竞争条件作为这种行为的原因。

情景 4：少数族裔区域中断

预期结果：Pod 被逐出无法访问的节点，而新 Pod 被调度到可用的符合条件的节点上。

少量中断意味着区域中与 Kubernetes 控制平面断开连接的节点比例较小。如果集群中未定义任何区域，则少数中断意味着整个集群中的少数节点已断开连接。如前所述，少数由节点生命周期控制器的`unhealthy-zone-threshold`设置来定义，默认情况下为55%。在这种情况下，如果网络断开的持续时间超过`default-unreachable-toleration-seconds`（5 分钟）和`node-monitor-grace-period`（40 秒），并且一个区域中只有不到 55% 的节点无法访问，则新 Pod 将调度到健康的节点上，而无法访问的节点上的 Pod 则被标记为驱逐。

集群管理员将看到在运行正常的节点上创建的新 Pod，断开连接的节点上的 pod 将显示为`Terminating`。请记住，尽管断开连接的节点上的 Pod 具有`Terminating`状态，但在节点重新连接到 Kubernetes 控制平面之前，它们不会被完全驱逐。

## 场景 5：网络中断期间节点重启

预期结果：无法访问的节点上的 Pod 要等到节点重新连接到 Kubernetes 控制平面后才会启动。Pod 故障转移遵循场景 1-3 中描述的逻辑，具体取决于无法访问的节点的数量。

在网络中断期间重启节点意味着在网络断开连接的同时，节点上发生了另一次故障（例如电源循环、内存不足事件或其他问题）。如果 kubelet 也已重新启动，则网络断开连接开始时在该节点上运行的 Pod 不会在断开连接期间自动重启。kubelet 在启动时会查询 Kubernetes API 服务器，以了解它应该

运行哪些 pod。如果 kubelet 由于网络断开连接而无法访问 API 服务器，则无法检索启动 pod 所需的信息。

在这种情况下，不能使用本地故障排除工具（例如 `crictl` CLI）来手动启动 pod，这是“漏洞”措施。Kubernetes 通常会移除失败的 pod 并创建新的 pod，而不是重启现有 pod（详情参见 containerd 存储库中的 #10213）。 GitHub 静态容器是唯一由 kubelet 控制的 Kubernetes 工作负载对象，可以在这些场景中重启。但是，通常不建议使用静态 pod 进行应用程序部署。相反，应在不同的主机上部署多个副本，以确保在同时出现多个故障（例如节点故障以及节点与 Kubernetes 控制平面之间的网络断开连接）时应用程序的可用性。

## 通过网络断开连接产生的应用程序网络流量

本页上的主题与 Kubernetes 集群网络以及节点与 Kubernetes 控制平面之间网络断开连接期间的应用程序流量有关。

### Cilium

Cilium 有多种模式用于 IP 地址管理 (IPAM)、封装、负载平衡和集群路由。本指南中验证的模式使用了集群范围 IPAM、VXLAN 叠加、BGP 负载平衡和 kube-proxy。Cilium 也在没有 BGP 负载平衡的情况下使用，取而代之的是 MetalL L2 负载平衡。

Cilium 安装的基础由 Cilium 操作员和 Cilium 代理组成。Cilium 运算符作为部署运行，注册 Cilium 自定义资源定义 (CRD)、管理 IPAM 以及将集群对象与 Kubernetes API 服务器同步等功能。Cilium 代理作为在每个节点上运行， DaemonSet 并管理 eBPF 程序，以控制集群上运行的工作负载的网络规则。

通常，在网络断开连接期间，Cilium 配置的集群内路由保持可用和原位，这可以通过观察集群内流量和 pod 网络的 IP 表 (iptables) 规则来确认。

```
ip route show table all | grep cilium
```

```
10.86.2.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.64/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.128/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.192/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.3.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16
10.86.3.16 dev cilium_host proto kernel scope link
...
```

但是，在网络断开连接期间，Cilium 操作员和 Cilium 代理会重新启动，因为他们的运行状况检查与与 Kubernetes API 服务器的连接的运行状况相结合。在网络断开连接期间，预计将在Cilium操作员和

Cilium代理的日志中看到以下内容。在网络断开连接期间，您可以使用诸如 `crictl` CLI 之类的工具来观察这些组件的重启情况，包括其日志。

```
msg="Started gops server" address="127.0.0.1:9890" subsys=gops
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Unable to contact k8s api-server" error="Get \"https://<k8s-cluster-ip>:443/
api/v1/namespaces/kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
 ipAddr="https://<k8s-cluster-ip>:443" subsys=k8s-client
msg="Start hook failed" function="client.(*compositeClientset).onStart
 (agent.infra.k8s-client)" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/
kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
msg="Start failed" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-
system\": dial tcp <k8s-cluster-ip>:443: i/o timeout" duration=1m5.003834026s
msg=Stopping
msg="Stopped gops server" address="127.0.0.1:9890" subsys=gops
msg="failed to start: Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-system
\": dial tcp <k8s-cluster-ip>:443: i/o timeout" subsys=daemon
```

如果您使用 Cilium 的 BGP 控制平面功能进行应用程序负载平衡，则在网络断开连接期间，Pod 和服务的 BGP 会话可能会关闭，因为 BGP 扬声器功能已与 Cilium 代理集成，而且 Cilium 代理在与 Kubernetes 控制平面断开连接时将持续重新启动。有关更多信息，请参阅 Cilium 文档中的 Cilium BGP 控制平面操作指南。此外，如果您在网络断开连接（例如电源循环或计算机重启）期间同时遇到故障，则不会通过这些操作保留 Cilium 路由，尽管当节点重新连接到 Kubernetes 控制平面并且 Cilium 再次启动时，会重新创建路由。

### Calico

即将推出

### MetalL B

MetalLB 有两种负载平衡模式：L2 模式和 BG P 模式。有关这些负载平衡模式的工作原理及其局限性的详细信息，请参阅 MetalLB 文档。本指南的验证使用了 L2 模式下的 MetalLB，在这种模式下，集群中的一台计算机拥有 Kubernetes 服务的所有权，并使用适用于 IPv4 的 ARP 使负载均衡器的 IP 地址可在本地网络上访问。运行MetalLB时，有一个控制器负责IP分配，每个节点上都有一个扬声器，负责通过分配的IP地址发布服务。MetalLB 控制器作为部署模式运行，MetalLB 扬声器作为部署模式运行。DaemonSet在网络断开连接期间，MetalLB 控制器和扬声器无法监视 Kubernetes API 服务器的集群资

源，但会继续运行。最重要的是，在网络断开连接期间，使用 MetalLB 进行外部连接的服务仍然可用且可访问。

## kube-proxy

在 EKS 集群中，kube-proxy 作为 DaemonSet 在每个节点上运行，负责管理网络规则，通过将服务 IP 地址转换为底层 Pod 的 IP 地址来实现服务与 Pod 之间的通信。kube-proxy 配置的 IP 表 (iptables) 规则在网络断开连接期间得到维护，集群内路由继续运行，kube-proxy pod 继续运行。

你可以使用以下 iptables 命令来观察 kube-proxy 规则。第一个命令显示通过PREROUTING链的数据包被定向到KUBE-SERVICES链。

```
iptables -t nat -L PREROUTING
```

```
Chain PREROUTING (policy ACCEPT)
target         prot opt source      destination
KUBE-SERVICES  all  --  anywhere    anywhere      /* kubernetes service portals */
```

通过检查KUBE-SERVICES链，我们可以看到各种集群服务的规则。

```
Chain KUBE-SERVICES (2 references)
target                     prot opt source      destination
KUBE-SVL-NZTS37XDTDNXGCKJ  tcp  --  anywhere    172.16.189.136  /* kube-system/hubble-
peer:peer-service cluster IP /
KUBE-SVC-2BINP2AXJOTI3HJ5  tcp  --  anywhere    172.16.62.72    / default/metallb-
webhook-service cluster IP /
KUBE-SVC-LRNEBRA3Z5YGJ4QC  tcp  --  anywhere    172.16.145.111  / default/redis-leader
 cluster IP /
KUBE-SVC-I7SKRZYQ7PWYV5X7  tcp  --  anywhere    172.16.142.147  / kube-system/eks-
extension-metrics-api:metrics-api cluster IP /
KUBE-SVC-JD5MR3NA4I4DYORP  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:metrics cluster IP /
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns cluster IP /
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns-tcp cluster IP /
KUBE-SVC-ENODL3HWJ5BZY56Q  tcp  --  anywhere    172.16.7.26     / default/frontend
 cluster IP /
KUBE-EXT-ENODL3HWJ5BZY56Q  tcp  --  anywhere    <LB-IP>    / default/frontend
 loadbalancer IP /
KUBE-SVC-NPX46M4PTMTKRN6Y  tcp  --  anywhere    172.16.0.1      / default/
kubernetes:https cluster IP /
```

```
KUBE-SVC-YU5RV2YQWHLZ5XPR  tcp  --  anywhere    172.16.228.76   / default/redis-
follower cluster IP /
KUBE-NODEPORTS             all  --  anywhere    anywhere        / kubernetes service
 nodeports; NOTE: this must be the last rule in this chain */
```

检查应用程序的前端服务链，我们可以看到支持该服务的 pod IP 地址。

```
iptables -t nat -L KUBE-SVC-ENODL3HWJ5BZY56Q
```

```
Chain KUBE-SVC-ENODL3HWJ5BZY56Q (2 references)
target                     prot opt source    destination
KUBE-SEP-EKXE7ASH7Y74BGBO  all  --  anywhere  anywhere    /* default/frontend ->
 10.86.2.103:80 / statistic mode random probability 0.33333333349
KUBE-SEP-GCY3OUXWSVMSEAR6  all  --  anywhere  anywhere    / default/frontend ->
 10.86.2.179:80 / statistic mode random probability 0.50000000000
KUBE-SEP-6GJJR3EF5AUP2WBU  all  --  anywhere  anywhere    / default/frontend ->
 10.86.3.47:80 */
```

在网络断开连接期间，当它尝试监视 Kubernetes API 服务器的节点和端点资源更新时，预计会出现以下 kube-proxy 日志消息。

```
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.Node: failed to list *v1.Node: Get \"https://<k8s-endpoint>/api/v1/nodes?
fieldSelector=metadata.name%3D<node-name>&resourceVersion=2241908\": dial tcp <k8s-
ip>:443: i/o timeout" logger="UnhandledError"
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get \"https://<k8s-endpoint>/
apis/discovery.k8s.io/v1/endpointslices?labelSelector=%21service.kubernetes.io
%2Fheadless%2C%21service.kubernetes.io%2Fservice-proxy-name&resourceVersion=2242090\":
 dial tcp <k8s-ip>:443: i/o timeout" logger="UnhandledError"
```

## CoreDNS

默认情况下，EKS 集群中的 Pod 使用 CoreDNS 集群 IP 地址作为集群内 DNS 查询的名称服务器。在 EKS 集群中，CoreDNS 作为节点上的部署运行。使用混合节点，当混合节点上有 CoreDNS 副本在本地运行时，Pod 可以在网络断开连接期间继续与 CoreDNS 通信。如果您的 EKS 集群的节点位于云中，而混合节点位于本地环境中，则建议每个环境中至少有一个 CoreDNS 副本。CoreDNS 继续为网络断开连接之前创建的记录提供 DNS 查询，并通过网络重新连接继续运行，以保持静态稳定。

在网络断开连接期间，当它尝试列出来自 Kubernetes API 服务器的对象时，预计会出现以下 CoreDNS 日志消息。

```
Failed to watch *v1.Namespace: failed to list *v1.Namespace: Get "https://<k8s-cluster-
ip>:443/api/v1/namespaces?resourceVersion=2263964": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.Service: failed to list *v1.Service: Get "https://<k8s-cluster-
ip>:443/api/v1/services?resourceVersion=2263966": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get "https://<k8s-
cluster-ip>:443/apis/discovery.k8s.io/v1/endpointslices?resourceVersion=2263896": dial
 tcp <k8s-cluster-ip>: i/o timeout
```

## 通过网络断开连接获得的主机凭证

EKS 混合节点与 AWS Systems Manager (SSM) 混合激活和 AWS IAM Anywhere 角色集成，用于通过 EKS 控制平面对节点进行身份验证的临时 IAM 证书。SSM 和 IAM Anywhere 角色都会自动刷新他们在本地主机上管理的临时证书。建议在集群中的混合节点上使用单一凭证提供商，即 SSM 混合激活或 IAM 角色随处可见，但不能两者兼而有之。

### SSM 混合激活

SSM 提供的临时证书有效期为一小时。使用 SSM 作为凭证提供商时，您无法更改凭证有效期。临时证书将在到期前由 SSM 自动轮换，并且轮换不会影响您的节点或应用程序的状态。但是，当 SSM 代理和 SSM 区域端点之间的网络断开连接时，SSM 无法刷新凭证，并且凭据可能会过期。

如果 SSM 无法连接到 SSM 区域端点，则使用指数退避来重试凭据刷新。在 SSM 代理版本`3.3.808.0`及更高版本（2024 年 8 月发布）中，指数级退缩上限为 30 分钟。根据网络断开连接的持续时间，SSM 最多可能需要 30 分钟才能刷新凭据，并且在刷新凭据之前，混合节点不会重新连接到 EKS 控制平面。在这种情况下，您可以重新启动 SSM 代理以强制刷新凭据。当前 SSM 凭证刷新行为的副作用是，节点可能会在不同的时间重新连接，具体取决于每个节点上的 SSM 代理何时设法刷新其凭据。因此，您可能会看到尚未重新连接的节点的 Pod 故障转移到已经重新连接的节点。

获取 SSM 代理版本。您也可以查看 SSM 控制台的舰队管理器部分：

```
# AL2023, RHEL
yum info amazon-ssm-agent
# Ubuntu
snap list amazon-ssm-agent
```

重新启动 SSM 代理：

```
# AL2023, RHEL
```

```
systemctl restart amazon-ssm-agent
# Ubuntu
systemctl restart snap.amazon-ssm-agent.amazon-ssm-agent
```

查看 SSM 代理日志：

```
tail -f /var/log/amazon/ssm/amazon-ssm-agent.log
```

网络断开连接期间的预期日志消息：

```
INFO [CredentialRefresher] Credentials ready
INFO [CredentialRefresher] Next credential rotation will be in 29.995040663666668
 minutes
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 35s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 56s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 1m24s before retrying retrieve credentials
```

## IAM Roles Anywhere

默认情况下，由 IAM Roles Anywhere 配置的临时证书有效期为一小时。您可以通过 IAM Roles Anywhere 配置文件中的`durationSeconds`字段配置凭证有效期。凭证的最长有效期为 12 小时。您的混合节点 IAM 角色的`MaxSessionDuration`设置必须大于您的 IAM Roles Anywhere 个人资料中的`durationSeconds`设置。

使用 IAM Rolet Anywhere 作为混合节点的凭证提供商时，网络断开连接后通常会在网络恢复后的几秒钟内重新连接到 EKS 控制平面，因为 kubelet 会调用按`aws_signing_helper credential-process`需获取证书。尽管与混合节点或网络断开连接没有直接关系，但在使用 IAM Roles Anywhere 时，您可以配置证书到期通知和警报。有关更多信息，请参阅 IAM 角色中的自定义 Anywhere 中的通知设置。

# 运行 AI/ML 工作负载的最佳实践

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

在 EKS 上运行 AI/ML 工作负载时实施最佳实践可以确保这些工作负载性能良好、经济实惠、弹性强且资源充足。最佳实践分为以下一般部分：计算、网络、存储、可观察性和性能。

## 反馈

本指南发布的目的是 GitHub 收集来自更广泛 EKS/Kubernetes 社区的直接反馈和建议。如果您认为我们应该在指南中包含最佳实践，请在 GitHub 存储库中提交问题或提交 PR。我们的意图是在服务中添加新功能或出现新的最佳实践时，定期更新指南。

## 计算和自动缩放

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

### GPU 资源优化和成本管理

#### 使用 Well-Known 标签安排具有 GPU 要求的工作负载

对于对不同 GPU 特性敏感 AI/ML 的工作负载（例如 GPU、GPU 内存），我们建议使用与 Karpenter 和托管节点组一起使用的节点类型支持的已知调度标签来指定 GPU 要求。未能定义这些可能会导致 Pod 被调度到 GPU 资源不足的实例上，从而导致故障或性能降低。我们建议使用 NodeSelector 或节点亲和性来指定 Pod 应在哪个节点上运行，并在 Pod 的资源部分设置计算资源（CPU、内存、GPU 等）。

示例

例如，在使用 Karpenter 时使用 GPU 名称节点选择器：

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod-example
spec:
  containers:
  - name: ml-workload
    image: <image>
    resources:
      limits:
        nvidia.com/gpu: 1  # Request one NVIDIA GPU
  nodeSelector:
    karpenter.k8s.aws/instance-gpu-name: "l40s"  # Run on nodes with NVIDIA L40S GPUs
```

## 使用 Kubernetes 设备插件公开 GPU

要在节点上公开 GPU，必须将 NVIDIA GPU 驱动程序安装在节点的操作系统上，并将容器运行时配置为允许 Kubernetes 调度器向具有可用 GPU 的节点分配 Pod。NVIDIA Kubernetes 设备插件的设置过程取决于你正在使用的 EKS 加速 AMI：

- Bottlerocket 加速的 AMI：这款 AMI 包含 NVIDIA GPU 驱动程序，NVIDIA Kubernetes 设备插件已预先安装并随时可用，开箱即用 GPU 支持。无需额外配置即可向 Kubernetes 调度器公开 GPU。
- AL2023 加速 AMI：此 AMI 包含 NVIDIA GPU 驱动程序，但未预装 NVIDIA Kubernetes 设备插件。 AL2023 您必须单独安装和配置设备插件，通常是通过 DaemonSet。请注意，如果您使用 eksctl 创建集群并在中指定 GPU 实例类型（例如`g5.xlarge`），则`eksctl`会自动选择相应的 ClusterConfig AMI 并安装 NVIDIA Kubernetes 设备插件。要了解更多信息，请参阅 eksctl 文档中的 GPU 支持。

如果你决定改用 EKS 加速 AMI 和 NVIDIA GPU 操作员来管理 NVIDIA Kubernetes 设备插件等组件，请注意根据 NVIDIA GPU 驱动程序和 NVIDIA 容器工具包 NVIDIA 文档禁用 NVIDIA GPU 驱动程序和 NVIDIA 容器工具包的管理 Pre-Installed NVIDIA Container Toolki t。

要验证 NVIDIA 设备插件是否处于活动状态以及 GPU 是否正确暴露，请运行：

```
kubectl describe node | grep nvidia.com/gpu
```

此命令检查`nvidia.com/gpu`资源是否在节点的容量和可分配的资源范围内。例如，应显示具有一个 GPU 的节点`nvidia.com/gpu: 1`。有关更多信息，请参阅 Kubernetes GPU 调度指南。

## 使用许多不同的 EC2 实例类型

如本the section called “数据层面”节所述，尽可能多地使用不同的 EC2 实例类型是在 Amazon EKS 上实现可扩展性的重要最佳实践。此建议也适用于采用加速硬件（例如 GPU）的实例。如果您创建的集群仅使用一种实例类型，并尝试将节点数量扩展到该区域的容量之外，则可能会收到容量不足错误 (ICE)，表示没有可用的实例。在进行任意多样化之前，了解 AI/ML 工作负载的独特特征非常重要。使用 EC2 Instance Type Explorer 工具查看可用的实例类型，生成符合您的特定计算要求的实例类型列表，并避免任意限制可在集群中使用的实例类型。

加速计算实例以不同的购买模式提供，以适应短期、中期和稳定状态的工作负载。对于短期、灵活且容错的工作负载，如果您希望避免进行预留，请查看竞价型实例。容量块、 On-Demand 实例和节省计划允许您为中长期工作负载持续时间预置加速计算实例。为了增加在首选购买选项中成功访问所需容量的机会，建议使用不同的实例类型和可用区列表。或者，如果您遇到特定购买模式的 ICE，请重试使用其他型号。

示例以下示例说明如何让 Karpenter NodePool 能够配置大于第 3 代（例如 p3）的 G 和 P 实例。要了解更多信息，请参阅可扩展性一节。

```
- key: karpenter.k8s.aws/instance-category
  operator: In
  values: ["g", "p"] # Diversifies across G-series and P-series
- key: karpenter.k8s.aws/instance-generation
  operator: Gt
  values: ["3"] # Selects instance generations greater than 3
```

有关在 GPU 上使用竞价型实例的详细信息，请参阅下文的 “考虑使用带有 Karpenter 的 GPU 的 Amazon EC2 竞价型实例”。

## 考虑在 Karpenter 上使用适用于 GPU 的 Amazon EC2 竞价型实例

Amazon EC2 竞价型实例允许您利用 AWS 云中未使用的 EC2 容量，与 On-Demand 价格相比，可享受高达 90% 的折扣。当 EC2 需要恢复容量时，可以在两分钟内中断 Amazon EC2 竞价型实例。有关更多信息，请参阅 Amazon EC2 用户指南中的竞价型实例。Amazon EC2 Spot 是容错、无状态和灵活（时间和实例类型）工作负载的绝佳选择。要详细了解何时使用竞价型实例，请参阅 EC2 竞价型实例最佳实践。如果是，您也可以将竞价型实例用于 AI/ML 工作负载 Spot-friendly。

使用案例

Spot-friendly 工作负载可以是大数据、容器化工作负载 CI/CD、无状态 Web 服务器、高性能计算 (HPC) 和渲染工作负载。Spot 实例不适用于不灵活、有状态、不容错或实例节点之间紧密耦合的工作负载（例如，具有并行进程的工作负载，这些进程严重依赖彼此进行计算，需要持续的节点间通信，例

如计算流体动力学等 MPI-based 高性能计算应用程序或具有复杂相互依赖关系的分布式数据库）。以下是我们推荐的具体用例（排名不分先后）：

- Real-time 在线推理：只要您的工作负载适合点位，就可使用竞价型实例对实时推理工作负载进行成本优化的扩展。换句话说，推理时间要么少于两分钟，要么应用程序对中断具有容错能力，并且可以在不同的实例类型上运行。通过实例多样性（例如，跨多个实例类型和可用区）或预留来确保高可用性，同时实施应用程序级容错以应对潜在的 Spot 中断。
- Hyper-parameter 调整：使用 Spot 实例机会主义地运行探索性调整作业，因为可以容忍中断而不会造成重大损失，特别是对于持续时间较短的实验。
- 数据增强：使用 Spot 实例执行数据预处理和增强任务，如果中断，这些任务可以从检查点重新启动，因此非常适合 Spot 的可变可用性。
- Fine-tuning 模型：使用竞价型实例通过强大的检查点机制进行微调，从上次保存的状态恢复，从而最大限度地减少实例中断的影响。
- B@@ atch In ference：使用 Spot 实例以非实时方式处理大批量的离线推理请求，从而可以暂停和恢复作业，从而与 Spot 节省的成本保持最佳一致，并通过重试或多样化处理潜在的中断。
- 机会主义训练子集：将 Spot 实例用于边缘或实验性训练工作负载（例如，低于 1000 万个参数的小型模型），在这些工作负载中，中断是可以接受的，并且可以应用效率优化，例如跨实例类型或区域的多样化，尽管由于潜在的中断，不建议将其用于生产规模的训练。

注意事项

要在 Amazon EKS 上使用竞价型实例处理加速工作负载，需要考虑一些关键因素（排名不分先后）：

- 使用 Karpenter 管理启用了高级整合的竞价型实例。通过指定木匠。 sh/capacity-在你的 Karpenter 中键入为 “spot” NodePool，Karpenter 将默认配置 Spot 实例，无需任何其他配置。但是，要启用高级 Spot-to-Spot 整合（用价格较低的竞价替代品取代未充分利用的竞价节点），您需要通过在 Karpenter 控制器参数中设置--fe at ure-gates SpotToSpotConsolidation =true 或通过 FEATURE_GATES 环境变量来启用功能门。 SpotToSpotConsolidation Karpenter 使用价格容量优化的分配策略来配置 EC2 实例。根据 NodePool 要求和容器限制，Karpenter 将不可调度的容器装入垃圾箱，并将一组不同的实例类型发送到 Amazon EC2 队列 API。您可以使用 EC2 实例类型浏览器工具生成符合您的特定计算要求的实例类型列表。
- 确保工作负载无状态、容错且灵活。工作负载必须是无状态的、容错的，而且大小必须灵活。instance/GPU 这允许在 Spot 中断后无缝恢复，实例灵活性使您有可能在 Spot 上停留更长的时间。使用 AWS SQS 队列的名称配置 settings.InterruptionQueue Helm 值，以捕获竞价中断事件，从而在 Karpenter 中启用 Spot 中断处理。例如，通过 Helm 安装时，使用--set “settings.interruptionQueue=$ {CLUSTER_NAME}”。要查看示例，请参阅 Karpenter 入门指南。当

Karpenter 注意到 Spot 中断事件时，它会在中断事件发生之前自动封锁、污染、排空和终止节点，以最大限度地延长 Pod 的终止宽限期。同时，Karpenter 将立即启动一个新节点，这样它就可以尽快准备就绪。

- 避免过度限制实例类型选择。应尽量避免限制实例类型。通过不限制实例类型，以更低的成本获得竞价型实例中断频率的大规模竞价型容量的几率更高。例如，避免仅限于特定类型（例如 g5.xlarge）。考虑使用诸如 karpenter.k8s 之类的密钥指定一组不同的实例类别和生成。 aws/instance-类别和 karpenter.k8s。 aws/instance-世代。Karpenter 可以更轻松地在多个实例类型和可用区 (AZ) 之间实现按需实例和竞价型实例容量的多样化。此外，如果您的 AI/ML 工作负载需要特定或有限数量的加速器，但在不同区域之间具有灵活性，则可以在启动前使用 Spot Placement Score 动态确定部署工作负载的最佳区域。
- 扩大 NodePool 要求范围以包括更多类似的 EC2 实例系列。每个竞价型实例池都包含特定可用区 (AZ) 中特定实例类型的未使用的 EC2 实例容量。当 Karpenter 尝试配置新节点时，它会选择符合其要求 NodePool的实例类型。如果任何可用区中都没有兼容的实例类型具有 Spot 容量，则配置将失败。为避免此问题，请允许来自NVIDIA的更广泛的 g 系列实例（第 4 代或更高版本），跨大小和可用区 (AZ)，同时考虑 GPU 内存或光线追踪等硬件需求。由于实例的类型可能不同，因此您需要确保您的工作负载能够在每种类型上运行，并且获得的性能可以满足您的需求。
- 利用一个区域中的所有可用区。可用容量因可用区 (AZ) 而异，特定实例类型可能在一个可用区不可用，但在另一个可用区却很多。实例类型和可用区域的每种唯一组合构成一个单独的竞价容量池。通过在符合您的 Karpenter NodePool 要求的区域内的所有可用区中请求容量，您实际上是在同时搜索更多存储池。这样可以最大限度地提高竞价容量池的数量，从而增加获取竞价容量的可能性。要实现此目的，请在 NodePool 配置中省略 topology.kubernetes。 io/zone 完全键入以允许 Karpenter 从该区域的所有可用区中进行选择，或者使用运算符:in 明确列出可用区并提供值（例如 us-west-2a）。
- 考虑使用竞价投放分数 (SPS) 来了解使用竞价型实例成功访问所需容量的可能性。竞价投放分数 (SPS) 是一种提供分数的工具，可帮助您评估竞价请求成功的可能性。使用 SPS 时，您首先要为竞价型实例指定计算要求，然后 Amazon EC2 会返回您的竞价请求可能成功的前 10 个区域或可用区 (AZ)。区域和可用区的评分范围为 1 至 10。分数为 10 表示您的竞价请求极有可能成功，但不能保证成功。分数为 1 则表示您的竞价请求几乎没有成功的可能。即使是不同的区域或可用区，也可能会返回相同的分数。要了解更多信息，请参阅在 AWS 上构建 Spot 投放分数追踪器控制面板的指南。由于 Spot 容量一直在波动，SPS 将帮助您确定哪种实例类型、可用区和区域组合最适合您的工作负载限制（即灵活性、性能、大小等）。如果您的 AI/ML 工作负载需要特定或有限数量的加速器，但在区域之间具有灵活性，则可以在启动前使用竞价放置分数来动态确定部署工作负载的最佳区域。为了帮助您自动找出获得 Spot 容量的可能性，我们提供了构建 SPS 跟踪器仪表板的指南。该解决方案使用 YAML 配置监控 SPS 分数以实现多样化设置（例如，包括 GPU 在内的实例需求），将指标存储在中 CloudWatch，并提供用于比较配置的仪表板。为每个工作负载定义控制面板以评估 vCPU、内存和 GPU 需求，确保为 EKS 集群提供最佳设置，包括考虑使用其他 AWS 区域。要了解更多信息，请参阅 Spot 展示位置分数的工作原理。

- 优雅地处理 Spot 中断并进行测试。对于终止时间超过两分钟的 Pod，旧节点将在重新调度 pod 之前中断，这可能会影响工作负载的可用性。在设计应用程序时，请考虑两分钟的 Spot 中断通知，在长时间运行的应用程序（例如，将进度保存到 Amazon S3 等永久存储）中断后恢复，延长 Pod 规范中的终止时间GracePeriodSeconds（默认为 30 秒）以留出更多时间进行优雅关闭，并在应用程序中使用 PreStop 生命周期挂钩 and/or SIGTERM 信号处理中断，以进行清理、状态保存和连接关闭等优雅关闭活动。对于实时工作负载，如果扩展时间很重要，而且工作负载需要超过两分钟的时间才能准备好为流量提供服务，可以考虑通过查看和the section called “性能”最佳实践来优化容器启动the section called “仓储服务”和机器学习模型加载时间。要测试替换节点，请使用 AWS 故障注入服务 (FIS) 来模拟 Spot 中断。

除了这些核心的 Spot 最佳实践外，在 Amazon EKS 上管理 GPU 工作负载时，还要考虑这些因素。与 CPU-based 工作负载不同，GPU 工作负载对硬件细节（例如 GPU 功能和可用的 GPU 内存）特别敏感。GPU 工作负载可能会受到其可以使用的实例类型的限制，与 CPU 相比，可用选项较少。首先，评估您的工作负载是否具有实例灵活性。如果您不知道您的工作负载可以使用多少实例类型，请分别对其进行测试以确保兼容性和功能性。确定在尽可能实现多元化方面的灵活性，同时确认多元化可以使工作负载正常运行，并了解任何性能影响（例如对吞吐量或完成时间）的影响。作为实现工作负载多样化的一部分，请考虑以下几点：

- 查看 CUDA 和框架兼容性。您的 GPU 工作负载可能会针对特定的硬件、GPU 类型（例如，p3 中的 V100 与 p4 中的 A100）进行优化，或者针对特定的 CUDA 版本编写 TensorFlow，例如库，因此请务必查看工作负载的兼容性。这种兼容性对于防止运行时错误、崩溃、GPU 加速失败（例如，CUDA 版本与类似 PyTorch 或 TensorFlow 可能阻止执行的框架不匹配）或利用精度等 FP16/INT8 硬件功能的能力至关重要。
- GPU 内存。请务必使用诸如 DCGM Exporter 之类的工具评估模型的内存需求并分析模型在运行时的内存使用情况，并在众所周知的标签（如 karpenter.k8s）中设置实例类型所需的最低 GPU 内存。 aws/instance-gpu-memory。GPU VRAM 因实例类型而异（例如，NVIDIA T4 有 16GB，A10G 有 24GB，V100 有 16-32GB），机器学习模型（例如大型语言模型）可能会超过可用内存，从而导致内存不足 (OOM) 错误或崩溃。对于 EKS 中的竞价型实例，这可能会限制多样化。例如，如果您的模型不适合，则不能包含较低的 VRAM 类型，这可能会限制对容量池的访问并增加中断风险。请注意，对于单个 GPU、单节点推理（例如，在同一节点上调度多个 Pod 以利用其 GPU 资源），这可能会限制多样化，因为您只能在 Spot 配置中包含具有充足 VRAM 的实例类型。
- Floating-point 精度和性能。并非所有 Nvidia GPU 架构都具有相同的浮点精度（例如 FP16/INT8）。评估您的工作负载所需的核心类型 (CUDA/Tensor/RT) 性能和浮点精度。在价格较低、性能较低的 GPU 上运行并不意味着性能更好，因此可以考虑根据在特定时间范围内完成的工作来评估性能，以了解多元化的影响。

场景：实时推理工作负载的多样化

对于竞价型实例的实时在线推理工作负载，您可以将 Karpenter NodePool 配置为在兼容的 GPU 实例系列和世代之间实现多样化。这种方法通过从多个 Spot 池中提取资源来确保高可用性，同时通过 GPU 功能、内存和架构的限制来保持性能。它支持在实例容量受限时使用替代方案，最大限度地减少中断并优化推理延迟。此示例 NodePool 说明使用大于 3 的 g 和 p 系列实例，它们的 GPU 内存超过 20GB。

示例

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference-spot
spec:
  template:
    metadata:
      labels:
        role: gpu-spot-worker
    spec:
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["spot"] # Use Spot Instances
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g", "p"] # Diversifies across G-series and P-series
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"] # Selects instance generations greater than 3
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"] # Specifies AMD64 architecture, compatible with NVIDIA GPUs
        - key: karpenter.k8s.aws/instance-gpu-memory
          operator: Gt
          values: ["20480"] # Ensures more than 20GB (20480 MiB) total GPU memory
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
      nodeClassRef:
        name: gpu-inference-ec2
        group: karpenter.k8s.aws
        kind: EC2NodeClass
```

```
      expireAfter: 720h
  limits:
    cpu: 100
    memory: 100Gi
  disruption:
    consolidationPolicy: WhenEmptyOrUnderutilized
    consolidateAfter: 5m # Enables consolidation of underutilized nodes after 5 minutes
```

## 为长时间运行的训练作业实施检查点

Checkpointing 是一种容错技术，它涉及定期保存进程的状态，允许它在出现中断时从上次保存的点恢复。在机器学习中，它通常与训练相关联，在这种训练中，长时间运行的作业可以保存模型权重和优化器状态，以便在出现故障（例如硬件问题或竞价型实例中断）后恢复训练。

在训练期间，您可以使用检查点来保存机器学习 (ML) 模型的状态。检查点是模型的快照，可以通过机器学习框架的回调函数进行配置。您可以使用保存的检查点，根据上次保存的检查点重新启动训练作业。由于训练作业或实例意外中断，您可以使用检查点保存训练中的模型快照。这允许您在将来从检查点恢复对模型的训练。除了实施节点弹性系统外，我们还建议实施检查点以减轻中断的影响，包括硬件故障或 Amazon EC2 竞价型实例中断造成的中断的影响。

如果没有检查点，中断可能会导致计算时间浪费和进度损失，这对于长时间运行的训练作业来说代价高昂。Checkpointing 允许作业定期保存其状态（例如模型权重和优化器状态），并在中断后从上一个检查点（上次处理的批次）恢复。要实现检查点，请将您的应用程序设计为大批量处理数据，并在训练作业进行时将中间结果保存到永久存储中，例如通过 M ountpoint for Amazon S3 CSI 驱动程序保存到 Amazon S3 存储桶。

使用案例

检查点在特定场景中特别有用，可以平衡容错能力和性能开销。在以下情况下，可以考虑使用检查点功能：

- Job 持续时间超过几个小时：对于长时间运行的训练作业（例如，对于小型模型，或者 days/weeks 对于具有数十亿个参数的大型基础模型，>1-2 小时），在这种作业中，由于中断而导致的进度损失是代价高昂的。较短的工作可能无法证明 I/O 开销是合理的。
- 对于 Spot 实例或硬件故障：在容易出现中断的环境中，例如 EC2 Spot（2 分钟通知）或硬件故障（例如 GPU 内存错误），检查点功能可以快速恢复，这使得Spot在容错工作负载中可以节省成本。
- 大规模分布式训练：对于使用加速器（例如 >100 GPU） hundreds/thousands 的设置，其中平均故障间隔时间随着规模的增加而线性缩短。用于 model/data 并行性以处理并行检查点访问并避免完全重启。

- Large-scale 资源需求高的模型：在 PB 级的 LLM 训练中，由于集群规模，故障不可避免；分层方法（瞬态每隔 5-30 分钟进行一次快速本地化，重大故障每小时耐用一次）可以优化恢复时间与效率。

## 使用 ML 容量块为 P 和 Trainium 实例提供容量保障

机器学习容量块允许您预留备受追捧的 GPU 实例，特别是 P 实例（例如 p6-b200、p5、p5e、p5e、p5en、p4d、p4de）和 Trainium 实例（例如 trn1、trn2），以便几乎立即启动或在将来的某个日期启动，以支持您的短时机器学习 (ML) 工作负载。这些预留空间非常适合确保模型训练和微调等计算密集型任务的容量。EC2 容量块定价包括预留费和操作系统费。要了解有关定价的更多信息，请参阅适用于 ML 的 EC2 容量块定价。

要在 Amazon EKS 上为 AI/ML 工作负载预留 GPU 以获得可预测的容量保证，我们建议使用机器学习容量块进行短期或使用On-Demand 容量预留 (ODC) 来保证通用容量。

- ODCR 允许您在特定可用区内预留 EC2 实例容量（例如 g5 或 p5 等 GPU 实例），从而确保可用性，即使在需求旺盛时也是如此。ODCR 没有长期承诺，但您需要为预留容量付费，无论是已用容量还是闲置容 On-Demand 量。在 EKS 中，K arpenter 和托管节点组等节点类型支持 ODCR。要在 Karpenter 中确定优先级 ODCR，请将配置为使用该字段 NodeClass 。`capacityReservationSelectorTerms`请参阅 Karpenter 文档。 NodePools
- 容量块是 GPU（例如 p5、p4d）或 Trainium（trn1、trn2）实例的专用预留机制，专为模型训练、微调或实验等短期机器学习工作负载而设计。您可以从将来的某个日期开始在定义的时间段（通常为 24 小时到 182 天）内预留容量，只需为预留的时间付费。它们是预付费的，需要对容量需求进行预先规划，并且不支持自动扩展，但它们托管在 EC2 UltraClusters 中以实现低延迟联网。他们仅在保留期内收费。要了解更多信息，请参阅查找和购买容量块，或者按照为机器学习创建带容量块的托管节点组中的说明开始使用容量块设置托管节点组。

通过 AWS 管理控制台预留容量，并将您的节点配置为使用 ML 容量块。根据工作负载计划计划预留并在暂存集群中进行测试。有关更多信息，请参阅容量块文档。

## 考虑一下 On-Demand G Amazon EC2 实例的 Amazon EC2 竞价或 On-Demand 容量预留 (ODCR)

对于 G Amazon EC2 实例，请考虑不同的购买选项 On-Demand，即 Amazon EC2 竞价型实例和 On-Demand 容量预留。ODCR 允许您在特定可用区域中预留 EC2 实例容量一段时间，即使在需求旺盛时也能确保可用性。与仅适用于 P 和 Trainium 实例的 ML 容量块不同，ODCR 可用于更广泛的实例类型，包括 G 实例，因此适用于需要不同 GPU 功能的工作负载，例如推理或图形。使用 Amazon EC2

竞价型实例时，能够在不同的实例类型、大小和可用区之间进行多样化是能够在竞价上停留更长时间的关键。

ODCR 没有长期承诺，但您需要为预留容量付费，无论是已用容量还是闲置容 On-Demand 量。可以创建 ODC 以供立即使用，也可以安排在将来的某个日期使用，这为容量规划提供了灵活性。在 Amazon EKS 中，Karpenter 和托管节点组等节点类型支持 ODCR。要在 Karpenter 中确定优先级 ODCR，请将配置为使用该字段 NodeClass 。`capacityReservationSelectorTerms`请参阅 Karpenter 文档。 NodePools 有关创建 ODCR（包括 CLI 命令）的更多信息，请参阅On-Demand 容量预留入门。

### 考虑其他加速实例类型和大小

选择适当的加速实例和大小对于在 Amazon EKS 上优化机器学习工作负载的性能和成本至关重要。例如，不同的 GPU 实例系列具有不同的性能和功能，例如 GPU 内存。为了帮助您选择性价比最高的选项，请在 “加速计算” 下的 EC2 实例类型页面中查看可用的 GPU 实例。评估多种实例类型和大小，以找到最适合您的特定工作负载要求的实例类型和大小。考虑 GPU 数量、内存和网络性能等因素。通过仔细选择正确的 GPU 实例类型和大小，您可以在 EKS 集群中实现更好的资源利用率和成本效益。

如果您在 EKS 节点中使用 GPU 实例，则默认情况下，它的 `nvidia-device-plugin-daemonset` pod 将位于`kube-system`命名空间中。要快速了解你是否充分利用了实例中的 GPU，你可以使用 nvidia-smi，如下所示：

```
kubectl exec nvidia-device-plugin-daemonset-xxxxx \
  -n kube-system -- nvidia-smi \
  --query-
gpu=index,power.draw,power.limit,temperature.gpu,utilization.gpu,utilization.memory,memory.free
 \
  --format=csv -l 5
```

- 如果`utilization.memory`接近 100%，则您的代码很可能受限于内存。这意味着 GPU（内存）已得到充分利用，但可能表明应研究进一步的性能优化。
- 如果接近 100%，这并不一定意味着 GPU 已得到充分利用。`utilization.gpu`更好的衡量标准是与的`power.draw`比率`power.limit`。如果此比率为 100% 或更高，则说明您的代码充分利用了 GPU 的计算容量。
- 该`-l 5`标志显示每 5 秒输出一次指标。对于单个 GPU 实例类型，则不需要索引查询标志。

要了解更多信息，请参阅 AWS 文档中的 GPU 实例。

## 使用 Time-Slicing、MIG 和分数 GPU 分配优化 GPU 资源分配

Kubernetes 中的静态资源限制（例如 CPU、内存、GPU 数量）可能会导致过度配置或利用不足，对于推理等动态工作负载尤其如此。 AI/ML 选择正确的 GPU 很重要。对于低容量或尖峰工作负载，时间分片允许多个工作负载通过共享其计算资源共享单个 GPU，从而有可能提高效率并减少浪费。GPU 共享可以通过不同的选项实现：

- 利用节点选择器/节点亲和力来影响调度：确保已配置的节点和 Pod 调度到适合工作负载的 GPU 上（例如）`karpenter.k8s.aws/instance-gpu-name: "a100"`
- Time-Slicing：调度工作负载以随着时间的推移共享 GPU 的计算资源，从而允许在不进行物理分区的情况下并发执行。这非常适合具有可变计算需求但可能缺乏内存隔离的工作负载。
- Multi-Instance GPU (MIG)：MIG 允许将单个 NVIDIA GPU 分成多个隔离的实例，并支持 NVIDIA Ampere（例如 A100 GPU）、NVIDIA Hopper（例如 H100 GPU）和 NVIDIA Blackwell（例如 Blackwell GPU）GPU。每个 MIG 实例都会获得专用的计算和内存资源，从而在多租户环境或需要资源保障的工作负载中实现资源共享，这使您可以优化 GPU 资源利用率，包括通过时间分段为具有不同批次大小的多个模型提供服务等场景。
- GPU 分配：使用基于软件的调度将 GPU 的部分计算或内存分配给工作负载，从而为动态工作负载提供灵活性。作为该 Run:ai 平台的一部分，NVIDIA KAI 调度器通过允许 pod 请求一部分 GPU 资源来实现这一点。

要在 EKS 中启用这些功能，您可以部署 NVIDIA 设备插件，该插件将 GPU 作为可调度资源公开，并支持时间分片和 MIG。要了解更多信息，请参阅 Kubernetes 中的 Time-Slicing GPU 以及使用 NVIDIA 时间切片和加速 EC2 实例在 Amazon EKS 上共享 GPU。

示例

例如，要使用 NVIDIA 设备插件启用时间切片，请执行以下操作：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin-config
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    sharing:
      timeSlicing:
```

```
        resources:
        - name: nvidia.com/gpu
          replicas: 4  # Allow 4 pods to share each GPU
```

示例

例如，要使用 KAI 调度器进行小部分 GPU 分配，请将其与 NVIDIA GPU 操作员一起部署，然后在 pod 规范中指定小部分 GPU 资源：

```
apiVersion: v1
kind: Pod
metadata:
  name: fractional-gpu-pod-example
  annotations:
    gpu-fraction: "0.5"  # Annotation for 50% GPU
  labels:
    runai/queue: "default"  # Required queue assignment
spec:
  containers:
  - name: ml-workload
    image: nvcr.io/nvidia/pytorch:25.04-py3
    resources:
      limits:
        nvidia.com/gpu: 1
  nodeSelector:
    nvidia.com/gpu: "true"
  schedulerName: kai-scheduler
```

# 节点弹性和训练 Job 管理

## 通过自动恢复实现节点运行状况检查

对于 Amazon EKS 上需要频繁进行节点间通信的分布式训练作业（例如跨多个节点的多 GPU 模型训练），GPU 或 EFA 故障等硬件问题可能会导致训练作业中断。这些中断可能导致训练进度损失和成本增加，对于依赖稳定硬件的长时间运行 AI/ML 的工作负载尤其如此。

为了帮助增强针对硬件故障（例如运行 GPU 工作负载的 EKS 集群中的 GPU 故障）的弹性，我们建议使用带自动修复功能的 EKS 节点监控代理或 Amazon SageMaker HyperPod。而带自动修复功能的 EKS 节点监控代理提供节点运行状况监控和使用标准 Kubernetes 机制进行自动修复等功能，同时 SageMaker HyperPod 还提供针对性的弹性和专为大规模机器学习训练设计的其他功能，例如深度运行状况检查和自动任务恢复。

- 带有节点自动修复功能的 EKS 节点监控代理通过读取日志和应用 NodeConditions（包括加速硬件的标准条件和特定条件）来持续监控节点的运行状况，以识别 GPU 或网络故障等问题。`Ready`当节点被认为不健康时，Node Auto Repair 会封锁该节点并将其替换为新节点。Pod 的重新调度和作业的重启依赖于标准的 Kubernetes 机制和作业的重启策略。
- SageMaker HyperPod深度运行状况检查和运行状况监控代理持续监控 GPU 和 Trainium-based 实例的运行状况。它专为 AI/ML 工作负载量身定制，使用标签（例如节点健康状态）来管理节点运行状况。当节点被认为运行状况不佳时， HyperPod 会触发自动更换故障硬件（例如 GPU）。默认情况下，它通过基本的运行状况检查来检测 EFA 的网络相关故障，并支持对中断的训练作业自动恢复，允许作业从最后一个检查点继续，从而最大限度地减少大规模机器学习任务的中断。

对于具有自动修复功能的 EKS 节点监控代理和使用 EFA 的 SageMaker HyperPod 集群，要监控远程直接内存访问 (RDMA) 错误和数据包丢失等 EFA-specific 指标，请确保已安装 AWS EF A 驱动程序。此外，我们建议部署CloudWatch 可观察性 Add-on或使用带有 Prometheus 和 Grafana 的 DCGM Exporter 等工具来监控 EFA、GPU 以及与其功能相关的特定指标。 SageMaker HyperPod

## 对中断敏感型工作负载禁用 Karpenter 整合

对于对中断敏感的工作负载，例如处理、大规模 AI/ML 预测任务或训练，我们建议调整 Karpenter 整合策略，以防止作业执行期间出现中断。Karpenter 的整合功能通过终止未充分利用的节点或用价格较低的替代节点来自动优化集群成本。但是，即使工作负载完全利用了 GPU，Karpenter 也可能会整合节点，前提是它发现了价格较低且大小适中、符合容器要求的实例类型，从而导致作业中断。

`WhenEmptyOrUnderutilized`整合策略可能会过早终止节点，从而导致更长的执行时间。例如，由于容器重新调度、数据重新加载，中断可能会延迟任务的恢复，这对于长时间运行的批量推理作业来说可能会付出高昂的代价。为了缓解这种情况，您可以将设置`consolidationPolicy`为`WhenEmpty`并配置`consolidateAfter`持续时间（例如 1 小时），以便在工作负载高峰期间保留节点。例如：

```
disruption:
  consolidationPolicy: WhenEmpty
  consolidateAfter: 60m
```

这种方法可以改善尖峰批量推理工作负载和其他对中断敏感的作业（例如实时在线推理数据处理或模型训练）的 pod 启动延迟，在这些作业中，中断成本大于节省的计算成本。Karpenter NodePool 干扰预算是管理 Karpenter 中断的另一项功能。有了预算，您可以确保在所选时间点中断的节点不超过一定数量 NodePool 的节点。您还可以使用中断预算来防止所有节点在特定时间（例如高峰时段）中断。要了解更多信息，请参阅 Karpenter 整合文档。

### 使用 ttl 自动执行 Clean-Up Kub SecondsAfterFinished ernetes 作业

我们建议在 Amazon EKS 中将 Kubernetes 任务设置`ttlSecondsAfterFinished`为自动删除已完成的任务对象。延迟任务对象会消耗群集资源，例如 API 服务器内存，并且仪表板混乱（例如 Grafana、Amazon），从而使监控复杂化。 CloudWatch例如，将 TTL 设置为 1 小时可确保任务在完成后不久即被删除，从而保持集群整洁。有关更多详细信息，请参阅自动清理已完成的作业。

### 配置 Low-Priority Job 抢占功能 Higher-Priority Jobs/workloads

对于 Amazon EKS 上的混合优先级 AI/ML 工作负载，您可以配置低优先级任务抢占以确保优先级较高的任务（例如实时推理）及时获得资源。如果没有抢占权，低优先级工作负载（例如批处理推理、数据处理）、非批处理服务（例如后台任务、cron 作业）或 CPU/memory-intensive 作业（例如 Web 服务）可能会通过占用节点来延迟关键 pod。抢占允许 Kubernetes 在高优先级 Pod 需要资源时驱逐低优先级 Pod，从而确保在具有 GPU、CPU 或内存的节点上高效分配资源。我们建议使用 Kubernetes `PriorityClass` 来分配优先级并控制驱`PodDisruptionBudget`逐行为。

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: low-priority
value: 100
---
spec:
  priorityClassName: low-priority
```

有关更多信息，请参阅 Kubernetes 优先级和抢占文档。

## 应用程序扩展和性能

### 使用 Karpenter 或静态节点为机器学习工作负载量身定制计算容量

为了确保在 Amazon EKS 上为机器学习 (ML) 工作流程提供经济高效且响应迅速的计算容量，我们建议根据工作负载的特征和成本承诺量身定制节点配置策略。以下是两种需要考虑的方法：使用 Karpenter 进行即时扩展，以及预留容量的静态节点组。

- Just-in-time 像 Karpenter 这样的数据平面缩放器：对于具有可变计算需求的动态机器学习工作流程（例如， GPU-based 推理后 CPU-based 绘图），我们建议使用像 Karpenter 这样的即时数据平面缩放器。

- 将@@ 静态节点组用于可预测的工作负载：对于可预测的稳定状态机器学习工作负载或使用预留实例时，EKS 托管节点组可以帮助确保预留容量得到充分配置和利用，从而最大限度地节省开支。这种方法非常适合通过 RI 或 ODCR 提交的特定实例类型。

示例

这是一个多元化 Karpenter 的示例 NodePool，它允许启动实例生成量大于三的 g Amazon EC2 实例。

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference
spec:
  template:
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand"]
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"]
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
  limits:
    cpu: "1000"
    memory: "4000Gi"
    nvidia.com/gpu: "10"  *# Limit the total number of GPUs to 10 for the NodePool*
  disruption:
    consolidationPolicy: WhenEmpty
    consolidateAfter: 60m
```

```
    expireAfter: 720h
```

示例

使用静态节点组处理训练工作负载的示例：

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: ml-cluster
  region: us-west-2
managedNodeGroups:
  - name: gpu-node-group
    instanceType: p4d.24xlarge
    minSize: 2
    maxSize: 2
    desiredCapacity: 2
    taints:
      - key: nvidia.com/gpu
        effect: NoSchedule
```

### 使用污点和容忍度来防止在加速实例上调度非加速工作负载

在 GPU 资源上调度非加速工作负载的计算效率不高，我们建议使用污点和容忍度来确保非加速工作负载 pod 不会调度到不合适的节点上。有关更多信息，请参阅 Kubernetes 文档。

### 基于模型性能进行缩放

对于推理工作负载，我们建议使用 Kubernetes Event-Driven Autoscaling (KEDA) 根据模型性能指标（例如推理请求或令牌吞吐量）进行扩展，并设置适当的冷却时间。静态扩展策略可能会过度配置资源或资源不足，从而影响成本和延迟。要了解更多信息，请参阅 KEDA 文档。

## 用于高级 GPU 管理的动态资源分配

动态资源分配 (DRA) 代表了 Kubernetes GPU 资源管理的一项根本进步。DRA 超越了传统的设备插件限制，实现了复杂的 GPU 共享、拓扑感知和跨节点资源协调。DRA 在 Amazon EKS 1.33 版本中可用，它通过提供以下内容来解决 AI/ML 工作负载中的关键挑战：

- Fine-grained GPU 分配
- 高级共享机制，例如 Multi-Process 服务 (MPS) 和 Multi-Instance GPU (MIG)
- 支持下一代硬件架构，包括 NVIDIA GB200 UltraServers

传统的 GPU 分配将 GPU 视为不透明的整数资源，从而造成严重的利用率不足（在生产集群中通常为 30-40%）。之所以发生这种情况，是因为即使只需要少量资源，工作负载也可以独占访问整个 GPU。DRA 通过引入结构化的声明式分配来改变这种模型，让 Kubernetes 调度器能够完全了解硬件特征和工作负载要求。这可以实现明智的放置决策和高效的资源共享。

## 使用 DRA 代替 NVIDIA 设备插件的优势

NVIDIA 设备插件（从版本开始`0.12.0`）支持 GPU 共享机制，包括时间切片、MPS 和 MIG。但是，DRA 可以解决一些架构限制。

NVIDIA 设备插件限制

- 静态配置：GPU 共享配置（时间切片副本和 MPS 设置）需要在整个集群范围内进行预配置。`ConfigMaps`这使得为不同的工作负载提供不同的共享策略变得困难。
- 精细选择有限：虽然设备插件通过节点标签公开 GPU 特性，但工作负载无法在调度决策中动态请求特定的 GPU 配置（内存大小和计算能力）。
- 没有跨节点资源协调：无法管理跨多个节点的分布式 GPU 资源，也无法表达复杂的拓扑要求，例如 NVIDIA GB200 等系统的 NVLink 域。
- 调度器限制：Kubernetes 调度器将 GPU 资源视为不透明的整数，从而限制了其做出拓扑感知决策或处理复杂资源依赖关系的能力。
- 配置复杂性：设置不同的共享策略需要多重`ConfigMaps`且谨慎的节点标记，这增加了操作的复杂性。

采用 DRA 的解决方案

- 动态资源选择：DRA 允许工作负载在请求时指定详细要求（GPU 内存、驱动程序版本和特定属性）`resourceclaims`。这样可以实现更灵活的资源匹配。
- 拓扑感知：通过结构化参数和设备选择器，DRA 可以处理复杂的需求，例如跨节点 GPU 通信和内存一致性互连。
- Cross-node 资源管理：`computeDomains`支持跨多个节点协调分布式 GPU 资源，这对于像 GB200 这样具有 IMEX 通道的系统至关重要。
- Workload-specific 配置：每种策略和配置都`ResourceClaim`指定了不同的共享策略和配置，允许对每个工作负载进行精细控制，而不是集群范围的设置。
- 增强的调度程序集成：DRA 为调度器提供详细的设备信息，并允许根据硬件拓扑和资源特征做出更明智的放置决策。

重要提示：DRA 并不能完全取代 NVIDIA 设备插件。NVIDIA DRA 驱动程序与设备插件配合使用，可提供增强功能。设备插件继续处理基本的 GPU 发现和管理，而 DRA 则增加了高级分配和调度功能。

## DRA 支持的实例及其功能

DRA 支持因 Amazon EC2 实例系列和 GPU 架构而异，如下表所示。

| 实例系列 | GPU 类型 | Time-slicing | MIG 支持 | MPS 支持 | IMEX 支持 | 使用案例 |
|---|---|---|---|---|---|---|
| G5 | NVIDIA A10G | 是 | 否 | 是 | 否 | 推理和图形工作负载 |
| G6 | NVIDIA L4 | 是 | 否 | 是 | 否 | AI 推理和视频处理 |
| G6e | 英伟达 L40S | 是 | 否 | 是 | 否 | 训练、推理和图形 |
| P4d/P4de | NVIDIA A100 | 支持 | 是 | 是 | 否 | Large-scale 培训和 HPC |
| P5 | NVIDIA H100 | 支持 | 是 | 是 | 否 | 基础模型训练 |
| P6 | 英伟达 B200 | 支持 | 是 | 是 | 否 | 十亿或万亿参数模型、分布式训练和推理 |
| p6e | 英伟达 GB200 | 支持 | 是 | 是 | 是 | 十亿或万亿参数模型、分布式训练和推理 |

以下是对表中每项功能的描述：

- Time-slicing：允许多个工作负载随着时间的推移共享 GPU 计算资源。
- Multi-Instance GPU (MIG)：用于创建隔离的 GPU 实例的 Hardware-level 分区。
- Multi-Process 服务 (MPS)：允许在单个 GPU 上并发执行多个 CUDA 进程。
- 节点间内存交换 (IMEX)：GB200 的跨节点 Memory-coherent 通信。 UltraServers

## 其他资源

有关 Kubernetes DRA 和 NVIDIA DRA 驱动程序的更多信息，请参阅以下资源： GitHub

- Kubernetes 动态资源分配
- 针对 DRA 的 Kubernetes 增强提案
- 适用于 GPU 的 NVIDIA DR
- NVIDIA DRA 示例和快速入门

## 为高级 GPU 管理设置动态资源分配

以下主题介绍如何为高级 GPU 管理设置动态资源分配 (DRA)。

先决条件

在 Amazon EKS 上实施 DRA 之前，请确保您的环境满足以下要求。

集群配置

- 运行版本`1.33`或更高版本的 Amazon EKS 集群
- Amazon EKS 托管节点组（目前仅支持 AL2023 和 Bottlerocket NVIDIA 优化了 AMI 的托管节点组，Karpenter 不支持 DRA）AL2023
- 具有适当实例类型的 NVIDIA GPU-enabled 工作节点

必需的组件

- NVIDIA 设备插件版本`0.17.1`或更高版本
- 英伟达 DRA 驱动程序版本`25.3.0`或更高版本

步骤 1：使用 eksctl 创建带有 DRA-enabled 节点组的集群

1. 创建名为`dra-eks-cluster.yaml`：的集群配置文件

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
```

```
  region: us-west-2
  version: '1.33'

managedNodeGroups:
- name: gpu-dra-nodes
  amiFamily: AmazonLinux2023
  instanceType: g6.12xlarge
  desiredCapacity: 2
  minSize: 1
  maxSize: 3

  labels:
    node-type: "gpu-dra"
    nvidia.com/gpu.present: "true"

  taints:
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule
```

2. 创建集群：

```
eksctl create cluster -f dra-eks-cluster.yaml
```

第 2 步：部署 NVIDIA 设备插件

部署 NVIDIA 设备插件以启用基本的 GPU 发现：

1. 添加 NVIDIA 设备插件 Helm 存储库：

```
helm repo add nvidia https://nvidia.github.io/k8s-device-plugin
helm repo update
```

2. 为设备插件创建自定义值：

```
cat <<EOF > nvidia-device-plugin-values.yaml
gfd:
  enabled: true
nfd:
  enabled: true
tolerations:
  - key: nvidia.com/gpu
```

```
    operator: Exists
    effect: NoSchedule
EOF
```

3. 安装 NVIDIA 设备插件：

```
helm install nvidia-device-plugin nvidia/nvidia-device-plugin \
 --namespace nvidia-device-plugin \
 --create-namespace \
 --version 0.17.1 \
 --values nvidia-device-plugin-values.yaml
```

步骤 3：部署 NVIDIA DRA 驱动程序 Helm 图

1. 为 DRA 驱动程序创建`dra-driver-values.yaml`值文件：

```
---
nvidiaDriverRoot: /

gpuResourcesEnabledOverride: true

resources:
  gpus:
    enabled: true
  computeDomains:
    enabled: true  # Enable for GB200 IMEX support

controller:
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule

kubeletPlugin:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "nvidia.com/gpu.present"
            operator: In
            values: ["true"]
```

```
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

2. 添加 NVIDIA NGC Helm 存储库：

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update
```

3. 安装 NVIDIA DRA 驱动程序：

```
helm install nvidia-dra-driver nvidia/nvidia-dra-driver-gpu \
 --version="25.3.0-rc.2" \
 --namespace nvidia-dra-driver \
 --create-namespace \
 --values dra-driver-values.yaml
```

步骤 4：验证 DRA 的安装

1. 验证 DRA API 资源是否可用：

```
kubectl api-resources | grep resource.k8s.io/v1beta1
```

预期的输出如下所示：

```
deviceclasses resource.k8s.io/v1beta1 false DeviceClass
resourceclaims resource.k8s.io/v1beta1 true ResourceClaim
resourceclaimtemplates resource.k8s.io/v1beta1 true ResourceClaimTemplate
resourceslices resource.k8s.io/v1beta1 false ResourceSlice
```

2. 检查可用的设备类别：

```
kubectl get deviceclasses
```

以下是预期输出的示例：

```
NAME                                        AGE
compute-domain-daemon.nvidia.com            4h39m
compute-domain-default-channel.nvidia.com   4h39m
```

```
gpu.nvidia.com                                  4h39m
mig.nvidia.com                                  4h39m
```

当新创建的 G6 GPU 实例加入启用了 DRA 的 Amazon EKS 集群时，会发生以下操作：

- NVIDIA DRA 驱动程序会自动发现 A10G GPU 并在该节点`resourceslices`上创建两个 GPU。
- 该`gpu.nvidia.com`切片注册物理 A10G GPU 设备及其规格（内存、计算能力等）。
- 由于 A10G 不支持 MIG 分区，因此该`compute-domain.nvidia.com`切片会创建一个代表 GPU 整个计算环境的单个计算域。
- 然后将`resourceslices`它们发布到 Kubernetes API 服务器，从而使 GPU 资源可用于调度。`resourceclaims`

  DRA 调度器现在可以智能地将此 GPU 分配给通过其请求 GPU 资源的 Pod`resourceclaimtemplates`，与传统的设备插件方法相比，资源管理更加灵活。这会自动发生，无需人工干预。DRA 驱动程序完成资源发现和注册过程后，该节点即可用于 GPU 工作负载。

  当你运行以下命令时：

  ```
  kubectl get resourceslices
  ```

  以下是预期输出的示例：

  ```
  NAME                                                        NODE
              DRIVER                          POOL                                  AGE
  ip-100-64-129-47.ec2.internal-compute-domain.nvidia.com-rwsts
   ip-100-64-129-47.ec2.internal    compute-domain.nvidia.com
   ip-100-64-129-47.ec2.internal    35m
  ip-100-64-129-47.ec2.internal-gpu.nvidia.com-6kndg
   ip-100-64-129-47.ec2.internal    gpu.nvidia.com
   ip-100-64-129-47.ec2.internal    35m
  ```

继续the section called “使用动态资源分配安排简单的 GPU 工作负载”。

## 使用动态资源分配安排简单的 GPU 工作负载

要使用动态资源分配 (DRA) 安排简单的 GPU 工作负载，请执行以下步骤。在继续操作之前，请确保你已经遵循了the section called “为高级 GPU 管理设置动态资源分配”。

1. 使用名ResourceClaimTemplate为以下文件创建 GPU 分配的基本配置basic-gpu-claim-template.yaml：

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: gpu-test1

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  namespace: gpu-test1
  name: single-gpu
spec:
  spec:
    devices:
      requests:
      - name: gpu
        deviceClassName: gpu.nvidia.com
```

2. 应用模板：

```
kubectl apply -f basic-gpu-claim-template.yaml
```

3. 验证状态：

```
kubectl get resourceclaimtemplates -n gpu-test1
```

下面是示例输出：

```
NAME         AGE
single-gpu   9m16s
```

4. 创建一个使用的 PodResourceClaimTemplate，其文件名为basic-gpu-pod.yaml：

```
---
apiVersion: v1
kind: Pod
metadata:
  namespace: gpu-test1
```

```
  name: gpu-pod
  labels:
    app: pod
spec:
  containers:
  - name: ctr0
    image: ubuntu:22.04
    command: ["bash", "-c"]
    args: ["nvidia-smi -L; trap 'exit 0' TERM; sleep 9999 & wait"]
    resources:
      claims:
      - name: gpu0
  resourceClaims:
  - name: gpu0
    resourceClaimTemplateName: single-gpu
  nodeSelector:
    NodeGroupType: gpu-dra
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: "nvidia.com/gpu"
    operator: "Exists"
    effect: "NoSchedule"
```

5. 应用并监控 Pod：

```
kubectl apply -f basic-gpu-pod.yaml
```

6. 检查 Pod 状态：

```
kubectl get pod -n gpu-test1
```

以下是预期输出示例：

```
NAME      READY   STATUS    RESTARTS   AGE
gpu-pod   1/1     Running   0          13m
```

7. 检查ResourceClaim状态：

```
kubectl get resourceclaims -n gpu-test1
```

以下是预期输出示例：

```
NAME                  STATE                AGE
gpu-pod-gpu0-l76cg    allocated,reserved   9m6s
```

8. 查看 Pod 日志以查看 GPU 信息：

```
kubectl logs gpu-pod -n gpu-test1
```

以下是预期输出示例：

```
GPU 0: NVIDIA L4 (UUID: GPU-da7c24d7-c7e3-ed3b-418c-bcecc32af7c5)
```

继续阅读the section called “采用动态资源分配的 GPU 优化技术”以了解使用 DRA 的更高级的 GPU 优化技术。

## 采用动态资源分配的 GPU 优化技术

现代 GPU 工作负载需要复杂的资源管理才能实现最佳利用率和成本效益。DRA 支持多种高级优化技术，可解决不同的用例和硬件功能：

- Time-slicing允许多个工作负载随着时间的推移共享 GPU 计算资源，因此非常适合偶尔使用 GPU 的推理工作负载。有关示例，请参阅the section called “通过时间切片优化 GPU 工作负载”。
- Multi-Process 服务 (MPS) 支持在单个 GPU 上并发执行多个 CUDA 进程，隔离效果比时间切片更好。有关示例，请参阅the section called “使用 MPS 优化 GPU 工作负”。
- Multi-Instance GPU (MIG) 提供硬件级分区，使用专用的计算和内存资源创建隔离的 GPU 实例。有关示例，请参阅the section called “使用 GPU 优化 GP Multi-Instance U 工作负”。
- 节点间内存交换 (IMEX) 支持跨节点的内存一致性通信，以便在 NVIDIA GB200 系统上进行分布式训练。有关示例，请参阅the section called “使用 GB200 P6e 实例通过 IMEX 优化 GPU 工作负载”。

这些技术可以显著提高资源利用率。Organizations报告说，通过优化的共享策略，GPU利用率从传统分配的30-40%提高到80-90%。技术的选择取决于工作负载特性、隔离要求和硬件功能。

### 通过时间切片优化 GPU 工作负载

Time-slicing 通过调度多个工作负载在同一个物理 GPU 上按顺序运行，使它们能够共享 GPU 计算资源。它非常适合偶尔使用 GPU 的推理工作负载。

请执行以下步骤。

1. 使用名为：的文件ResourceClaimTemplate为时间切片定义一个 timeslicing-claim-template.yaml

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: timeslicing-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: timeslicing-gpu-template
  namespace: timeslicing-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: TimeSlicing
```

2. 使用时间切片定义一个 Pod，文件名为：timeslicing-pod.yaml

```
---
# Pod 1 - Inference workload
apiVersion: v1
kind: Pod
metadata:
  name: inference-pod-1
  namespace: timeslicing-gpu
  labels:
    app: gpu-inference
spec:
```

```
  restartPolicy: Never
  containers:
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 1 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate inference workload
          for i in range(20):
              x = torch.randn(1000, 1000).cuda()
              y = torch.mm(x, x.t())
              print(f"Pod 1 - Iteration {i+1} completed at {time.strftime('%H:%M:
%S')}")
              time.sleep(60)
      else:
          print("No GPU available!")
          time.sleep(5)
    resources:
      claims:
      - name: shared-gpu-claim
  resourceClaims:
  - name: shared-gpu-claim
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule


---
```

```
# Pod 2 - Training workload
apiVersion: v1
kind: Pod
metadata:
  name: training-pod-2
  namespace: timeslicing-gpu
  labels:
    app: gpu-training
spec:
  restartPolicy: Never
  containers:
  - name: training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 2 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate training workload with heavier compute
          for i in range(15):
              x = torch.randn(2000, 2000).cuda()
              y = torch.mm(x, x.t())
              loss = torch.sum(y)
              print(f"Pod 2 - Training step {i+1}, Loss: {loss.item():.2f} at
 {time.strftime('%H:%M:%S')}")
              time.sleep(5)
      else:
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim-2
  resourceClaims:
  - name: shared-gpu-claim-2
    resourceClaimTemplateName: timeslicing-gpu-template
```

```
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

3. 应用模板和 Pod：

```
kubectl apply -f timeslicing-claim-template.yaml
kubectl apply -f timeslicing-pod.yaml
```

4. 监控资源声明：

```
kubectl get resourceclaims -n timeslicing-gpu -w
```

下面是示例输出：

```
NAME                                        STATE                AGE
inference-pod-1-shared-gpu-claim-9p97x      allocated,reserved   21s
training-pod-2-shared-gpu-claim-2-qghnb     pending              21s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     pending              105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     allocated,reserved   105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
```

First Pod (`inference-pod-1`)

- 州：`allocated,reserved`
- 含义：DRA 找到了可用的 GPU 并将其保留给这个 Pod
- Pod 状态：立即开始运行

第二个吊舱 (`training-pod-2`)

- 州：`pending`
- 含义：等待 DRA 在同一 GPU 上配置时间切片
- Pod 状态：等待调度

- 该州将从pending变allocated,reserved为 running

使用 MPS 优化 GPU 工作负

Multi-Process 服务 (MPS) 支持在单个 GPU 上并发执行多个 CUDA 上下文，隔离效果比时间切片更好。

请执行以下步骤。

1. 使用名为mps-claim-template.yaml：的文件为 MPS 定义 ResourceClaimTemplate

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: mps-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mps-gpu-template
  namespace: mps-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: MPS
```

2. 使用 MPS 定义一个 Pod，文件名为mps-pod.yaml：

```
---
# Single Pod with Multiple Containers sharing GPU via MPS
```

```
apiVersion: v1
kind: Pod
metadata:
  name: mps-multi-container-pod
  namespace: mps-gpu
  labels:
    app: mps-demo
spec:
  restartPolicy: Never
  containers:
  # Container 1 - Inference workload
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import torch.nn as nn
      import time
      import os

      print(f"=== INFERENCE CONTAINER STARTING ===")
      print(f"Process ID: {os.getpid()}")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")

      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

          # Create inference model
          model = nn.Sequential(
              nn.Linear(1000, 500),
              nn.ReLU(),
              nn.Linear(500, 100)
          ).cuda()

          # Run inference
          for i in range(1, 999999):
              with torch.no_grad():
                  x = torch.randn(128, 1000).cuda()
                  output = model(x)
```

```
                result = torch.sum(output)
                print(f"Inference Container PID {os.getpid()}: Batch {i}, Result:
{result.item():.2f} at {time.strftime('%H:%M:%S')}")
            time.sleep(2)
    else:
        print("No GPU available!")
        time.sleep(60)
  resources:
    claims:
    - name: shared-gpu-claim
      request: shared-gpu

 # Container 2 - Training workload
 - name: training-container
   image: nvcr.io/nvidia/pytorch:25.04-py3
   command: ["python", "-c"]
   args:
   - |
     import torch
     import torch.nn as nn
     import time
     import os

     print(f"=== TRAINING CONTAINER STARTING ===")
     print(f"Process ID: {os.getpid()}")
     print(f"GPU available: {torch.cuda.is_available()}")
     print(f"GPU count: {torch.cuda.device_count()}")

     if torch.cuda.is_available():
         device = torch.cuda.current_device()
         print(f"Current GPU: {torch.cuda.get_device_name(device)}")
         print(f"GPU Memory:
{torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

         # Create training model
         model = nn.Sequential(
             nn.Linear(2000, 1000),
             nn.ReLU(),
             nn.Linear(1000, 500),
             nn.ReLU(),
             nn.Linear(500, 10)
         ).cuda()

         criterion = nn.MSELoss()
```

```
          optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

          # Run training
          for epoch in range(1, 999999):
              x = torch.randn(64, 2000).cuda()
              target = torch.randn(64, 10).cuda()

              optimizer.zero_grad()
              output = model(x)
              loss = criterion(output, target)
              loss.backward()
              optimizer.step()

              print(f"Training Container PID {os.getpid()}: Epoch {epoch}, Loss:
{loss.item():.4f} at {time.strftime('%H:%M:%S')}")
              time.sleep(3)
      else:
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim
        request: shared-gpu

 resourceClaims:
 - name: shared-gpu-claim
   resourceClaimTemplateName: mps-gpu-template

 nodeSelector:
   NodeGroupType: "gpu-dra"
   nvidia.com/gpu.present: "true"
 tolerations:
 - key: nvidia.com/gpu
   operator: Exists
   effect: NoSchedule
```

3. 应用模板并创建多个 MPS 容器：

```
kubectl apply -f mps-claim-template.yaml
kubectl apply -f mps-pod.yaml
```

4. 监控资源声明：

```
kubectl get resourceclaims -n mps-gpu -w
```

下面是示例输出：

```
NAME                                                STATE                AGE
mps-multi-container-pod-shared-gpu-claim-2p9kx   allocated,reserved   86s
```

此配置演示了使用 NVIDIA Multi-Process 服务 (MPS) 通过动态资源分配 (DRA) 实现真正的 GPU 共享。与按顺序轮流使用 GPU 的工作负载的时间切片不同，MPS 允许两个容器在同一个物理 GPU 上同时运行。关键见解是，DRA MPS 共享需要在单个 Pod 中使用多个容器，而不是多个单独的 Pod。部署后，DRA 驱动程序会向 Pod 分配一个ResourceClaim，并自动配置 MPS 以允许推理容器和训练容器同时执行。

每个容器都有自己独立的 GPU 内存空间和计算资源，MPS 守护程序会协调对底层硬件的访问。您可以通过执行以下操作来验证它是否有效：

- 勾选nvidia-smi，这会将两个容器显示为 M+C (MPS + Compute) 进程共享同一 GPU 设备。
- 监控来自两个容器的日志，这将显示交错的时间戳，证明可以同时执行。

这种方法允许互补的工作负载高效共享昂贵的 GPU 硬件，而不是让单个进程充分利用它，从而最大限度地提高 GPU 利用率。

**## 1#####**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:09:30 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
```

```
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   48C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+-----------------------------------------------------------------------------------------
+
| Processes:
   |
|  GPU   GI   CI              PID   Type   Process name                        GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=========================================================================================|
|    0   N/A  N/A               1    M+C   python
 246MiB |
+-----------------------------------------------------------------------------------------
+
```

**## 2#####**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:16:00 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
```

```
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   51C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+---------------------------------------------------------------------------------------
+
| Processes:
   |
|  GPU   GI   CI              PID   Type   Process name                        GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=======================================================================================|
|    0   N/A  N/A               1    M+C   python
 314MiB |
+---------------------------------------------------------------------------------------
+
```

使用 GPU 优化 GP Multi-Instance U 工作负

Multi-instance GPU (MIG) 提供硬件级分区，使用专用的计算和内存资源创建隔离的 GPU 实例。

使用具有各种配置文件的动态 MIG 分区需要 NVIDIA GPU 操作员。NVIDIA GPU 操作员使用 MIG Manager 创建 MIG 配置文件并重新启动 P4D、P4De、P5、P6 等 GPU 实例以应用配置更改。GPU 操作员通过 MIG Manager 组件包括全面的 MIG 管理功能，该组件可监视节点标签的变化并自动应用相应的 MIG 配置。当请求更改 MIG 配置文件时，操作员会优雅地关闭所有 GPU 客户端，应用新的分区几何结构，然后重新启动受影响的服务。此过程需要重启 GPU 实例的节点，以确保干净的 GPU 状态转换。这就是为什么`WITH_REBOOT=true`在 MIG 管理器配置中启用对于成功部署 MIG 至关重要。

你需要 NVIDIA DRA 驱动程序和 NVIDIA GPU 操作员才能在 Amazon EKS 中使用 MIG。除此之外，你不需要 NVIDIA 设备插件和 DCGM Exporter，因为它们是 NVIDIA GPU 操作员的一部分。由于 EKS NVIDIA AMI 预装了 NVIDIA 驱动程序，因此我们禁用了 GPU 操作员部署驱动程序，以避免冲突并利用实例上已经存在的优化驱动程序。NVIDIA DRA 驱动程序处理 MIG 实例的动态资源分配，而 GPU 操作员则管理整个 GPU 生命周期。这包括 MIG 配置、设备插件功能、通过 DCGM 进行监控以及节点

功能发现。这种集成方法为企业 GPU 管理提供了完整的解决方案，具有硬件级隔离和动态资源分配功能。

步骤 1：部署 NVIDIA GPU 操作员

1. 添加 NVIDIA GPU 操作员存储库：

```
helm repo add nvidia https://nvidia.github.io/gpu-operator
helm repo update
```

2. 创建 gpu-operator-values.yaml 文件：

```
driver:
  enabled: false

mig:
  strategy: mixed

migManager:
  enabled: true
  env:
    - name: WITH_REBOOT
      value: "true"
  config:
    create: true
    name: custom-mig-parted-configs
    default: "all-disabled"
    data:
      config.yaml: |-
        version: v1
        mig-configs:
          all-disabled:
            - devices: all
              mig-enabled: false

          # P4D profiles (A100 40GB)
          p4d-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.5gb": 2
                "2g.10gb": 1
                "3g.20gb": 1
```

```
            - devices: [4, 5, 6, 7]
              mig-enabled: false

          # P4DE profiles (A100 80GB)
          p4de-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.10gb": 2
                "2g.20gb": 1
                "3g.40gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

devicePlugin:
  enabled: true
  config:
    name: ""
    create: false
    default: ""

toolkit:
  enabled: true

nfd:
  enabled: true

gfd:
  enabled: true

dcgmExporter:
  enabled: true
  serviceMonitor:
    enabled: true
    interval: 15s
    honorLabels: false
    additionalLabels:
      release: kube-prometheus-stack

nodeStatusExporter:
  enabled: false

operator:
  defaultRuntime: containerd
```

```
  runtimeClass: nvidia
  resources:
    limits:
      cpu: 500m
      memory: 350Mi
    requests:
      cpu: 200m
      memory: 100Mi

daemonsets:
  tolerations:
    - key: "nvidia.com/gpu"
      operator: "Exists"
      effect: "NoSchedule"
  nodeSelector:
    accelerator: nvidia
  priorityClassName: system-node-critical
```

3. 使用以下gpu-operator-values.yaml文件安装 GPU 操作员：

```
helm install gpu-operator nvidia/gpu-operator \
  --namespace gpu-operator \
  --create-namespace \
  --version v25.3.1 \
  --values gpu-operator-values.yaml
```

此 Helm 图表部署了以下组件和多个 MIG 配置文件：

- 设备插件（GPU 资源调度）
- DCGM 导出器（GPU 指标和监控）
- 节点功能发现（NFD-硬件标签）
- GPU 功能发现（GFD- GPU-specific 标签）
- MIG 管理器（Multi-instance GPU 分区）
- 容器工具包（GPU 容器运行时）
- 操作员控制器（生命周期管理）

4. 验证部署 Pod：

```
kubectl get pods -n gpu-operator
```

下面是示例输出：

```
NAME                                                              READY   STATUS
 RESTARTS        AGE
gpu-feature-discovery-27rdq                                       1/1     Running
 0               3h31m
gpu-operator-555774698d-48brn                                     1/1     Running
 0               4h8m
nvidia-container-toolkit-daemonset-sxmh9                          1/1     Running
 1 (3h32m ago)   4h1m
nvidia-cuda-validator-qb77g                                       0/1     Completed
 0               3h31m
nvidia-dcgm-exporter-cvzd7                                        1/1     Running
 0               3h31m
nvidia-device-plugin-daemonset-5ljm5                              1/1     Running
 0               3h31m
nvidia-gpu-operator-node-feature-discovery-gc-67f66fc557-q5wkt    1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-master-5d8ffddcsl6s6   1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-6t4w7           1/1     Running
 1 (3h32m ago)   4h1m
nvidia-gpu-operator-node-feature-discovery-worker-9w7g8           1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-k5fgs           1/1     Running
 0               4h8m
nvidia-mig-manager-zvf54                                          1/1     Running
 1 (3h32m ago)   3h35m
```

5. 创建带有 P4de 托管节点组的 Amazon EKS 集群，用于测试 MIG 示例：

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
  region: us-east-1
  version: '1.33'

managedNodeGroups:
# P4DE MIG Node Group with Capacity Block Reservation
- name: p4de-mig-nodes
  amiFamily: AmazonLinux2023
  instanceType: p4de.24xlarge
```

```
# Capacity settings
desiredCapacity: 0
minSize: 0
maxSize: 1

# Use specific subnet in us-east-1b for capacity reservation
subnets:
  - us-east-1b

# AL2023 NodeConfig for RAID0 local storage only
nodeadmConfig:
  apiVersion: node.eks.aws/v1alpha1
  kind: NodeConfig
  spec:
    instance:
      localStorage:
        strategy: RAID0

# Node labels for MIG configuration
labels:
  nvidia.com/gpu.present: "true"
  nvidia.com/gpu.product: "A100-SXM4-80GB"
  nvidia.com/mig.config: "p4de-half-balanced"
  node-type: "p4de"
  vpc.amazonaws.com/efa.present: "true"
  accelerator: "nvidia"

# Node taints
taints:
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule

# EFA support
efaEnabled: true

# Placement group for high-performance networking
placementGroup:
  groupName: p4de-placement-group
  strategy: cluster

# Capacity Block Reservation (CBR)
# Ensure CBR ID matches the subnet AZ with the Nodegroup subnet
```

```
  spot: false
  capacityReservation:
    capacityReservationTarget:
      capacityReservationId: "cr-abcdefghij"  # Replace with your capacity
 reservation ID
```

NVIDIA GPU Operator 使用添加到节点的标签，nvidia.com/mig.config: "p4de-half-balanced"并使用给定的配置文件对 GPU 进行分区。

6. 登录实p4de例。
7. 运行以下命令：

```
nvidia-smi -L
```

您应该会看到以下示例输出：

```
[root@ip-100-64-173-145 bin]# nvidia-smi -L
GPU 0: NVIDIA A100-SXM4-80GB (UUID: GPU-ab52e33c-be48-38f2-119e-b62b9935925a)
  MIG 3g.40gb     Device  0: (UUID: MIG-da972af8-a20a-5f51-849f-bc0439f7970e)
  MIG 2g.20gb     Device  1: (UUID: MIG-7f9768b7-11a6-5de9-a8aa-e9c424400da4)
  MIG 1g.10gb     Device  2: (UUID: MIG-498adad6-6cf7-53af-9d1a-10cfd1fa53b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-3f55ef65-1991-571a-ac50-0dbf50d80c5a)
GPU 1: NVIDIA A100-SXM4-80GB (UUID: GPU-0eabeccc-7498-c282-0ac7-d3c09f6af0c8)
  MIG 3g.40gb     Device  0: (UUID: MIG-80543849-ea3b-595b-b162-847568fe6e0e)
  MIG 2g.20gb     Device  1: (UUID: MIG-3af1958f-fac4-59f1-8477-9f8d08c55029)
  MIG 1g.10gb     Device  2: (UUID: MIG-401088d2-716f-527b-a970-b1fc7a4ac6b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-8c56c75e-5141-501c-8f43-8cf22f422569)
GPU 2: NVIDIA A100-SXM4-80GB (UUID: GPU-1c7a1289-243f-7872-a35c-1d2d8af22dd0)
  MIG 3g.40gb     Device  0: (UUID: MIG-e9b44486-09fc-591a-b904-0d378caf2276)
  MIG 2g.20gb     Device  1: (UUID: MIG-ded93941-9f64-56a3-a9b1-a129c6edf6e4)
  MIG 1g.10gb     Device  2: (UUID: MIG-6c317d83-a078-5c25-9fa3-c8308b379aa1)
  MIG 1g.10gb     Device  3: (UUID: MIG-2b070d39-d4e9-5b11-bda6-e903372e3d08)
GPU 3: NVIDIA A100-SXM4-80GB (UUID: GPU-9a6250e2-5c59-10b7-2da8-b61d8a937233)
  MIG 3g.40gb     Device  0: (UUID: MIG-20e3cd87-7a57-5f1b-82e7-97b14ab1a5aa)
  MIG 2g.20gb     Device  1: (UUID: MIG-04430354-1575-5b42-95f4-bda6901f1ace)
  MIG 1g.10gb     Device  2: (UUID: MIG-d62ec8b6-e097-5e99-a60c-abf8eb906f91)
  MIG 1g.10gb     Device  3: (UUID: MIG-fce20069-2baa-5dd4-988a-cead08348ada)
GPU 4: NVIDIA A100-SXM4-80GB (UUID: GPU-5d09daf0-c2eb-75fd-3919-7ad8fafa5f86)
GPU 5: NVIDIA A100-SXM4-80GB (UUID: GPU-99194e04-ab2a-b519-4793-81cb2e8e9179)
GPU 6: NVIDIA A100-SXM4-80GB (UUID: GPU-c1a1910f-465a-e16f-5af1-c6aafe499cd6)
GPU 7: NVIDIA A100-SXM4-80GB (UUID: GPU-c2cfafbc-fd6e-2679-e955-2a9e09377f78)
```

NVIDIA GPU Operator 已成功`p4de-half-balanced`将 MIG 配置文件应用于你的 P4DE 实例，按照配置创建了硬件级 GPU 分区。以下是分区的工作原理：

GPU 操作员应用了您的嵌入式 MIG 配置文件中的以下配置：

```
p4de-half-balanced:
  - devices: [0, 1, 2, 3]        # First 4 GPUs: MIG enabled
    mig-enabled: true
    mig-devices:
      "1g.10gb": 2               # 2x small instances (10GB each)
      "2g.20gb": 1               # 1x medium instance (20GB)
      "3g.40gb": 1               # 1x large instance (40GB)
  - devices: [4, 5, 6, 7]        # Last 4 GPUs: Full GPUs
    mig-enabled: false
```

根据你的`nvidia-smi -L`输出，GPU 操作员创建的内容如下：

- MIG-enabled GPU (0-3)：硬件已分区
  - GPU 0：英伟达 A100-SXM4-80GB
    - MIG 3g.40GB 设备 0 — 大型工作负载（40GB 内存，42 SM）
    - MIG 2g.20GB 设备 1 — 中型工作负载（20GB 内存，28 个 SM）
    - MIG 1g.10GB 设备 2 — 小型工作负载（10GB 内存，14 个 SM）
    - MIG 1g.10GB 设备 3 — 小型工作负载（10GB 内存，14 个 SM）
  - GPU 1：英伟达 A100-SXM4-80GB
    - MIG 3g.40GB 设备 0 — 分区布局相同
    - MIG 2g.20GB 设备 1
    - MIG 1g.10GB 设备 2
    - MIG 1g.10GB 设备 3
  - GPU 2 和 GPU 3 — 与 GPU 0 和 GPU 1 的模式相同
- 完整 GPU (4-7)：没有 MIG 分区
  - GPU 4：NVIDIA A100-SXM4-80GB — 全 80GB GPU
  - GPU 5：NVIDIA A100-SXM4-80GB — 全 80GB GPU
  - GPU 6：NVIDIA A100-SXM4-80GB — 全 80GB GPU
  - GPU 7：NVIDIA A100-SXM4-80GB — 全 80GB GPU

NVIDIA GPU 操作员创建 MIG 分区后，NVIDIA DRA 驱动程序会自动检测这些硬件隔离的实例，并使其可用于在 Kubernetes 中进行动态资源分配。DRA 驱动程序会发现每个 MIG 实例及其特定配置文件（1g.10GB、2g.20GB、3g.40GB），并通过设备类别将其作为可调度资源公开。`mig.nvidia.com`

DRA 驱动程序持续监控 MIG 拓扑，并维护所有 GPU 上的可用实例清单。当 Pod 通过请求特定的 MIG 配置文件时`ResourceClaimTemplate`，DRA 驱动程序会智能地从任何可用的 GPU 中选择适当的 MIG 实例，从而实现真正的硬件级多租户。这种动态分配允许多个独立的工作负载在同一个物理 GPU 上同时运行，同时保持严格的资源边界和性能保证。

步骤 2：测试 MIG 资源分配

现在，让我们举一些示例，演示 DRA 如何动态地将 MIG 实例分配给不同的工作负载。部署`resourceclaimtemplates`并测试 pod，查看 DRA 驱动程序如何在可用的 MIG 分区中放置工作负载，从而允许多个容器在硬件级隔离下共享 GPU 资源。

1. 创建`mig-claim-template.yaml`以包含 MIG：`resourceclaimtemplates`

```
apiVersion: v1
kind: Namespace
metadata:
  name: mig-gpu

---
# Template for 3g.40gb MIG instance (Large training)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-large-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-large
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '3g.40gb'

---
# Template for 2g.20gb MIG instance (Medium training)
```

```
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-medium-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-medium
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '2g.20gb'

---
# Template for 1g.10gb MIG instance (Small inference)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-small-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-small
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '1g.10gb'
```

2. 应用三个模板：

```
kubectl apply -f mig-claim-template.yaml
```

3. 运行以下命令：

```
kubectl get resourceclaimtemplates -n mig-gpu
```

下面是示例输出：

```
NAME                  AGE
mig-large-template    71m
mig-medium-template   71m
mig-small-template    71m
```

4. 创建mig-pod.yaml以安排多个作业以利用这一点resourceclaimtemplates：

```
---
# ConfigMap containing Python scripts for MIG pods
apiVersion: v1
kind: ConfigMap
metadata:
  name: mig-scripts-configmap
  namespace: mig-gpu
data:
  large-training-script.py: |
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import time
    import os

    print(f"=== LARGE TRAINING POD (3g.40gb) ===")
    print(f"Process ID: {os.getpid()}")
    print(f"GPU available: {torch.cuda.is_available()}")
    print(f"GPU count: {torch.cuda.device_count()}")

    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
 1e9:.1f} GB")

        # Large model for 3g.40gb instance
        model = nn.Sequential(
            nn.Linear(2048, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
```

```
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Large batch for 3g.40gb
            x = torch.randn(256, 2048).cuda()
            y = torch.randint(0, 10, (256,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Large Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(3)

        print("Large training completed on 3g.40gb MIG instance")

 medium-training-script.py: |
   import torch
   import torch.nn as nn
   import torch.optim as optim
   import time
   import os

   print(f"=== MEDIUM TRAINING POD (2g.20gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")

   if torch.cuda.is_available():
       device = torch.cuda.current_device()
       print(f"Using GPU: {torch.cuda.get_device_name(device)}")
       print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
1e9:.1f} GB")
```

```
        # Medium model for 2g.20gb instance
        model = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Medium batch for 2g.20gb
            x = torch.randn(128, 1024).cuda()
            y = torch.randint(0, 10, (128,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Medium Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(4)

        print("Medium training completed on 2g.20gb MIG instance")

 small-inference-script.py: |
   import torch
   import torch.nn as nn
   import time
   import os

   print(f"=== SMALL INFERENCE POD (1g.10gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")
```

```
    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
 1e9:.1f} GB")

        # Small model for 1g.10gb instance
        model = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Inference loop
        for i in range(200):
            with torch.no_grad():
                # Small batch for 1g.10gb
                x = torch.randn(32, 512).cuda()
                output = model(x)
                prediction = torch.argmax(output, dim=1)

                if i % 20 == 0:
                    print(f"Small Inference - Batch {i}, Predictions:
 {prediction[:5].tolist()}, GPU Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(2)

        print("Small inference completed on 1g.10gb MIG instance")

---
# Pod 1: Large training workload (3g.40gb)
apiVersion: v1
kind: Pod
metadata:
  name: mig-large-training-pod
  namespace: mig-gpu
  labels:
    app: mig-large-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: large-training-container
```

```
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/large-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-large-claim
  resourceClaims:
  - name: mig-large-claim
    resourceClaimTemplateName: mig-large-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 2: Medium training workload (2g.20gb) - can run on SAME GPU as Pod 1
apiVersion: v1
kind: Pod
metadata:
  name: mig-medium-training-pod
  namespace: mig-gpu
  labels:
    app: mig-medium-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: medium-training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/medium-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
```

```
      readOnly: true
    resources:
      claims:
      - name: mig-medium-claim
  resourceClaims:
  - name: mig-medium-claim
    resourceClaimTemplateName: mig-medium-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 3: Small inference workload (1g.10gb) - can run on SAME GPU as Pod 1 & 2
apiVersion: v1
kind: Pod
metadata:
  name: mig-small-inference-pod
  namespace: mig-gpu
  labels:
    app: mig-small-inference
    workload-type: inference
spec:
  restartPolicy: Never
  containers:
  - name: small-inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/small-inference-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-small-claim
  resourceClaims:
```

```
  - name: mig-small-claim
    resourceClaimTemplateName: mig-small-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755
```

5. 应用这个规范，它应该部署三个 Pod：

```
kubctl apply -f mig-pod.yaml
```

这些 Pod 应由 DRA 驱动程序调度。

6. 查看 DRA 驱动程序 Pod 日志，您将看到类似于以下内容的输出：

```
I0717 21:50:22.925811 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:22.932499 1 driver.go:129] Returning newly prepared devices
 for claim '933e9c72-6fd6-49c5-933c-a896407dc6d1': [&Device{RequestNames:
[mig-large],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-0-
mig-9-4-4,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-0-mig-9-4-4**],}]
I0717 21:50:23.186472 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.191226 1 driver.go:129] Returning newly prepared devices
 for claim '61e5ddd2-8c2e-4c19-93ae-d317fecb44a4': [&Device{RequestNames:
[mig-medium],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-2-
mig-14-0-2,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-2-mig-14-0-2**],}]
I0717 21:50:23.450024 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.455991 1 driver.go:129] Returning newly prepared devices
 for claim '1eda9b2c-2ea6-401e-96d0-90e9b3c111b5': [&Device{RequestNames:
[mig-small],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-1-
mig-19-2-1,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-1-mig-19-2-1**],}]
```

7. 验证`resourceclaims`以查看 Pod 状态：

```
kubectl get resourceclaims -n mig-gpu -w
```

下面是示例输出：

```
NAME                                                STATE                AGE
mig-large-training-pod-mig-large-claim-6dpn8     pending              0s
mig-large-training-pod-mig-large-claim-6dpn8     pending              0s
mig-large-training-pod-mig-large-claim-6dpn8     allocated,reserved   0s
mig-medium-training-pod-mig-medium-claim-bk596   pending              0s
mig-medium-training-pod-mig-medium-claim-bk596   pending              0s
mig-medium-training-pod-mig-medium-claim-bk596   allocated,reserved   0s
mig-small-inference-pod-mig-small-claim-d2t58    pending              0s
mig-small-inference-pod-mig-small-claim-d2t58    pending              0s
mig-small-inference-pod-mig-small-claim-d2t58    allocated,reserved   0s
```

如你所见，所有 Pod 都从 “待处理” 移至 `allocated,reserved` DRA 驱动程序。

8. `nvidia-smi`从节点运行。你会注意到有三个 Python 处理器正在运行：

```
root@ip-100-64-173-145 bin]# nvidia-smi
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01 Driver Version: 570.158.01 CUDA Version: 12.8 |
|-----------------------------------------+------------------------
+----------------------+
| GPU Name Persistence-M | Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap | Memory-Usage | GPU-Util Compute M. |
| | | MIG M. |
|=========================================+========================
+======================|
| 0 NVIDIA A100-SXM4-80GB On | 00000000:10:1C.0 Off | On |
| N/A 63C P0 127W / 400W | 569MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 1 NVIDIA A100-SXM4-80GB On | 00000000:10:1D.0 Off | On |
| N/A 56C P0 121W / 400W | 374MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 2 NVIDIA A100-SXM4-80GB On | 00000000:20:1C.0 Off | On |
| N/A 63C P0 128W / 400W | 467MiB / 81920MiB | N/A Default |
```

```
| | | Enabled |
+-----------------------------------------+-----------------------
+----------------------+
| 3 NVIDIA A100-SXM4-80GB On | 00000000:20:1D.0 Off | On |
| N/A 57C P0 118W / 400W | 249MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+-----------------------
+----------------------+
| 4 NVIDIA A100-SXM4-80GB On | 00000000:90:1C.0 Off | 0 |
| N/A 51C P0 77W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 5 NVIDIA A100-SXM4-80GB On | 00000000:90:1D.0 Off | 0 |
| N/A 46C P0 69W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 6 NVIDIA A100-SXM4-80GB On | 00000000:A0:1C.0 Off | 0 |
| N/A 52C P0 74W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 7 NVIDIA A100-SXM4-80GB On | 00000000:A0:1D.0 Off | 0 |
| N/A 47C P0 72W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+



+-----------------------------------------------------------------------------------------
+
| MIG devices: |
+------------------+--------------------------------+-----------
+----------------------+
| GPU GI CI MIG | Memory-Usage | Vol| Shared |
| ID ID Dev | BAR1-Usage | SM Unc| CE ENC DEC OFA JPG |
| | | ECC| |
|==================+================================+===========
+======================|
| 0 2 0 0 | 428MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 2MiB / 32767MiB | | |
+------------------+--------------------------------+-----------
+----------------------+
```

```
| 0 3 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 0 9 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 0 10 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 1 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 1 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 1 13 0 2 | 161MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 2MiB / 8191MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 1 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 2 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 2 5 0 1 | 289MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 2MiB / 16383MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 2 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
| 2 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+-----------------+----------------------------------+-----------
+----------------------+
```

```
| 3 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 3 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+



+-----------------------------------------------------------------------------------------
+
| Processes: |
| GPU GI CI PID Type Process name GPU Memory |
| ID ID Usage |
|
=========================================================================================|
**| 0 2 0 64080 C python 312MiB |
| 1 13 0 64085 C python 118MiB |
| 2 5 0 64073 C python 210MiB |**
+-----------------------------------------------------------------------------------------
+
```

使用 GB200 P6e 实例通过 IMEX 优化 GPU 工作负载

IMEX（节点间内存交换）支持跨节点的内存一致性通信，以便在 NVIDIA GB200 上进行分布式训练。UltraServers

请执行以下步骤。

1. 使用名ComputeDomainimex-compute-domain.yaml为以下的文件为多节点训练定义一个

```
apiVersion: resource.nvidia.com/v1beta1
kind: ComputeDomain
```

```
metadata:
  name: distributed-training-domain
  namespace: default
spec:
  numNodes: 2
  channel:
    resourceClaimTemplate:
      name: imex-channel-template
```

2. 使用 IMEX 通道定义一个 Pod，文件名为：`imex-pod.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: imex-distributed-training
  namespace: default
  labels:
    app: imex-training
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: nvidia.com/gpu.clique
            operator: Exists
  containers:
  - name: distributed-training
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["bash", "-c"]
    args:
    - |
      echo "=== IMEX Channel Verification ==="
      ls -la /dev/nvidia-caps-imex-channels/
      echo ""

      echo "=== GPU Information ==="
      nvidia-smi
      echo ""

      echo "=== NCCL Test (if available) ==="
      python -c "
      import torch
      import torch.distributed as dist
```

```
      import os

      print(f'CUDA available: {torch.cuda.is_available()}')
      print(f'CUDA device count: {torch.cuda.device_count()}')

      if torch.cuda.is_available():
          for i in range(torch.cuda.device_count()):
              print(f'GPU {i}: {torch.cuda.get_device_name(i)}')

      # Check for IMEX environment variables
      imex_vars = [k for k in os.environ.keys() if 'IMEX' in k or 'NVLINK' in k]
      if imex_vars:
          print('IMEX Environment Variables:')
          for var in imex_vars:
              print(f'  {var}={os.environ[var]}')

      print('IMEX channel verification completed')
      "

      # Keep container running for inspection
      sleep 3600
    resources:
      claims:
      - name: imex-channel-0
      - name: imex-channel-1
  resourceClaims:
  - name: imex-channel-0
    resourceClaimTemplateName: imex-channel-template
  - name: imex-channel-1
    resourceClaimTemplateName: imex-channel-template
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

> **Note**
>
> 这需要 P6e GB200 实例。

3. 通过应用ComputeDomain和模板部署 IMEX：

```
kubectl apply -f imex-claim-template.yaml
```

```
kubectl apply -f imex-compute-domain.yaml
kubectl apply -f imex-pod.yaml
```

4. 检查ComputeDomain状态。

```
kubectl get computedomain distributed-training-domain
```

5. 监控 IMEX 守护程序的部署。

```
kubectl get pods -n nvidia-dra-driver -l resource.nvidia.com/computeDomain
```

6. 在 Pod 中查看 IMEX 频道：

```
kubectl exec imex-distributed-training -- ls -la /dev/nvidia-caps-imex-channels/
```

7. 查看 Pod 日志：

```
kubectl logs imex-distributed-training
```

以下是预期输出的示例：

```
=== IMEX Channel Verification ===
total 0
drwxr-xr-x. 2 root root 80 Jul 8 10:45 .
drwxr-xr-x. 6 root root 380 Jul 8 10:45 ..
crw-rw-rw-. 1 root root 241, 0 Jul 8 10:45 channel0
crw-rw-rw-. 1 root root 241, 1 Jul 8 10:45 channel1
```

有关更多信息，请参阅上的 NVIDIA 示例 GitHub。

# CPU 推理和编排

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

# 概述

CPU 实例是 Amazon EKS 上各种 AI 工作负载的一流计算选项。从小型语言模型 (SLM) 和经典的机器学习推理到数据管道和代理编排，CPU 提供了强大的性价比、广泛的容量可用性和熟悉的 Kubernetes 调度语义。

CPU 和 GPU 是互补的，没有竞争力。随着代理人工智能管道的复杂性增加，CPU 工作负载也随之增长：每个推理调用都被工具执行、上下文汇编、矢量搜索、护栏和编排逻辑所包围，所有这些都在 CPU 上运行。我们建议在设计架构时刻意使用两种计算类型，将每种工作负载放在性价比最高的层上。

并非每个工作负载都需要 GPU。路由、分类、检索、嵌入、编排以及越来越多的语言模型推理都在 CPU 上有效运行。 Current-generation 跨越arm64和x86的CPU实例为机器学习推理提供了强大的性价比。再加上 Karpenter 的节点整合、KEDA 的事件驱动扩展和量化模型服务，这提供了一个生产就绪型堆栈，平台团队无需深厚的 GPU 专业知识即可运行该堆栈。

本指南适用于：

- 平台工程师为 AI 工作负载设计多租户 EKS 集群。
- 机器学习从业者评估300B参数下的模型的推理后端。
- FinOps 在不牺牲 SLO 的情况下寻找具体的成本杠杆的@@ 团队。

你将学到什么：

- 哪些 AI 工作负载属于 CPU，哪些地方需要 GPU 或 Trainium。
- 如何将四维决策框架应用于任何新的工作负载。
- 两种生产模式：代理SLM预过滤和高密度模型农场。
- 优化最佳实践：量化、装箱、Spot 调度和自动缩放。

> **Warning**
>
> 本指南中的每一项建议都应根据经验进行验证。正确的实例系列（arm64、x86、GPU 或 Trainium）取决于您的模型、数据和延迟预算。使用本指南作为明智的起点，然后在提交之前进行基准测试。

# 为什么 CPU 用于 AI 工作负载？

生产 AI 管道将工作分散到多个计算层。CPU 处理路由、分类、检索、编排和越来越多的推理。Current-generation CPU 实例具有很高的性价比和熟悉的 Kubernetes 调度，使其成为许多 AI 工作负载的实用选择。

有三个因素使 CPU 对这些工作负载具有吸引力：

## 容量可用性

GPU 实例通常需要提前几周预留容量。CPU 实例可在所有 AWS 地区广泛使用，没有专门的设备插件，没有 DRA 配置，也没有 MIG 分区。当您需要快速扩展时，CPU 容量是最容易获得的选择。

## 经济学

Current-generation CPU 实例为机器学习推理提供了强劲的性价比。对于运行 FinOps 审查或管理多租户集群的团队来说，CPU 和 GPU 之间的成本差异很大，特别是对于量化的 SLM 而言，GPU 加速带来的回报会越来越少。我们建议对可用实例系列（Graviton、AMD、Intel）进行基准测试，以找到最适合您的工作负载的每代币成本。

## 操作简单

CPU 容器使用标准的 Kubernetes 调度（`requests`、、节点亲和性`limits`、拓扑分布）。没有设备插件，没有自定义调度器，没有`nvidia.com/gpu`资源类型。想要在没有深厚的 GPU 专业知识的情况下运行 AI 工作负载的团队可以更快地使用 CPU 实现生产。

## 代理管道中的 CPU 表面越来越大

在代理人工智能管道中，每个 GPU 推理调用都被 CPU 工作所包围：工具执行、上下文组装、矢量搜索、嵌入查询、护栏、响应验证、内存管理和编排逻辑。随着代理变得越来越复杂（更多的工具、更长的链、多步推理），这些 CPU 工作负载会以超线性方式增长。像 MCP（模型上下文协议）这样的协议进一步放大了这一点：每个 MCP 工具调用都会触发完全在 CPU 上运行的数据检索、转换和格式化。

# CPU 与 GPU /Trainium：何时选择每个

| 因素 | 选择 CPU | 选择 GPU/Trainium |
| --- | --- | --- |
| 模型大小 | SLM 1-8B（量化）；嵌入；分类器 | 8B+ 用于低延迟在线推理；一般为 70B + |

| 因素 | 选择 CPU | 选择 GPU/Trainium |
| --- | --- | --- |
| 延迟 SLO | p95 100-500ms 可以接受 | p95 需要 < 50 毫秒 |
| 并发 | req/s 每个端点 < 100 | > 100 req/s 持续 |
| 工作负载类型 | 编排、检索、ETL、批量评分 | 在线推理、微调、训练 |
| Capacity | 即时可用，无需预订 | 通常需要预留容量 |
| 成本敏感度 | CPU 为符合条件的工作负载提供最优惠的美元/代币 | GPU 在高利用率时摊销 |
| 团队专业知识 | 标准的 Kubernetes 操作 | 需要 GPU 操作知识 |
| 数据主权 | VPC 中的 SLM 推理；完整的审计跟踪；数据永远不会离开您的账户 | 自行管理则相同；不适用于外部 API |

> **Tip**
>
> 这些阈值是起点。我们建议在提交计算层之前，使用实际模型和流量模式在候选实例系列（arm64 和 x86）上运行目标推理引擎。

## 工作负载决策框架

为 AI 工作负载选择正确的计算可归结为四个维度：

1. 模型大小和精度：量化能否将质量保持在可接受的范围内？
2. 延迟和吞吐量 SL O：您的 p50/p95 目标和峰值请求速率是什么？
3. 工作负载类型：在线推理、批量评分、检索还是编排？
4. 成本和容量限制： FinOps 预算、区域供应情况、预订策略？

使用下表作为决策矩阵。

| 工作负载 | CPU | GPU/Trainium | 注意 |
| --- | --- | --- | --- |
| SLM（1-8B 参数，量化） | 默认选择。延迟 100-500 毫秒时性价比高，QPS 适中。跨实例系列进行基准测试。 | 当 p95 <50ms or concurrency >100 req/s 的时候。 | 推荐 Q4_K_M 或 Q8_0 量化 |
| 中型模型（8-30B 参数） | Batch、异步、离线评分。测试第 4 季度的量化。 | 在线推理、较长的上下文、极短的延迟。 | 跨实例系列的第 4 季度基准测试 |
| 大型 LLM（70B 以上的参数） | Non-real-time 只是，大量量化 | 生产在线推理的默认值 | 即使是 70B 也能在 CPU 上运行；预计延迟会很高 |
| 古典机器学习/嵌入/简历 | High-density 服务；跨节点装箱。 | 大视野或大规模多模式。 | TorchServe，CPU 上的 Triton 可以处理数千个模型。 |
| 数据管道/ETL/合成数据 | CPU 上的 Ray 和 Spark 用于数据准备和功能工程。 | N/A | CPU 支撑了整个数据准备阶段 |
| 代理编排/RAG 检索 | Network-bound 服务：API 网关、代理层、检索器、区块器。 | N/A | 受益于高带宽 CPU 实例。 |
| Fine-tuning /训练 | 数据准备和管道编排。 | 模型训练和提炼。 | 混合动力：CPU 准备、GPU 训练、CPU 推断。 |
| Compliance-sensitive 推断（金融服务业、医疗保健、政府） | CPU 上的 VPC 中的 SLM。数据保留在账户中，完整审计跟踪。 | 如果在 GPU 上进行自我管理，则相同。 | 在受监管的环境中，CPU 凭借低于 8B 的机型的成本获胜。 |

> **Warning**
>
> 虽然从技术上讲，在大量量化的 CPU 上运行 70B 以上的模型是可能的（第四季度或更低），但这仅适用于非实时、离线或批处理工作负载。预计代币生成率将保持在较低的个位数（1-5 tokens/sec），即使在第四季度，内存需求也将超过40GB，对于较长的输出，每次响应的延迟以分钟为单位。对于任何交互式或对延迟敏感的用例，700亿以上的模型属于GPU或Trainium。

### 快速基准测试工作流程

在提交实例系列之前，我们建议您运行结构化基准测试，将候选的 CPU 系列（arm64 和 x86）与 GPU 进行比较，并使用一个可比指标：目标 p95 延迟下的每 1,000 次查询成本。每个系列部署一个具有相同模型配置（相同的量化、上下文大小、线程数）的节点，对每个节点进行负载测试并进行比较。如果 CPU 实例符合你的 p95 SLO，它很可能会在成本上获胜。如果漏掉的幅度很小，请在迁移到 GPU 之前尝试该系列中的最新一代产品。如果您的并发目标延迟仍然过高，那就是将工作负载转移到 GPU 的信号。

## 生产模式

### 模式 1：Agentic AI — CPU Pre-Filter 上的 SLM 和 LLM 升级

大多数代理工作流程都会重复执行相同的狭窄模式：对请求进行分类、选择工具、提取结构化数据、验证响应。这些任务不需要 70B 参数模型。

对 SLM (ar Xiv:2506.02153) 的研究表明，在 100B 参数下的模型，当专门用于某个域时，可以在受限的子任务上匹配或超过大型 LLM，同时以显著降低的成本和延迟在 CPU 上高效运行。当模型针对特定领域进行微调时，其较小的占用空间可以使其比调用通用 LLM 更准确、更便宜。

在这种模式下，CPU 上的 SLM 可以端到端地处理大多数请求。路由层（也在 CPU 上运行）只能将真正复杂的情况升级为 GPU-hosted LLM。

在 CPU 上运行的组件：

- API 网关/代理层 — 处理身份验证、路由、速率限制
- 代理协调器-管理工具调用和状态
- SLM 推理服务 — 在 CPU 上使用推理引擎（例如 llama.cpp、Ollama 或 vLLM）进行量化的 1-8B 模型
- 向量检索- OpenSearch 在 CPU 节点上

开启的组件 GPU/Trainium：

- 大型法学硕士，用于复杂综合、长上下文推理

为什么这种模式起作用：在许多代理工作流程中，60-80% 的请求可由 SLM 分类或提取。对于您避免的每个 LLM 调用，您还可以避免周围的 CPU 工作，例如组装大型上下文窗口、在长响应中运行护栏以及管理复杂状态。CPU 层独立于 GPU 层进行扩展。

典型代理管道中的 CPU 工作负载类别包括：工具执行（MCP 服务器调用、API 调用、数据库查询）、上下文汇编、矢量搜索和嵌入查询、编排和规划逻辑、护栏和安全筛选、响应验证和格式化、代理内存和状态管理，以及。 logging/observability

这种模式也适合微调生命周期：在 CPU 节点上收集域数据，在 GPU 上进行微调，然后将量化模型部署回 CPU 进行推理，成本比 LLM 端点低得多。LoRa Land（ar Xiv:2405.00732）的研究表明，经过微调的7B模型 GPT-4 在测试的大多数特定领域任务中表现优于其他领域，这使得许多生产用例都可行“微调小型模型并在CPU上运行” 路径。

### 模式 2： High-Density CPU 模型群

生产机器学习管道通常会部署成百上千个较小的模型：嵌入模型、推荐模型、分类器、 BERT-based 评分器和计算机视觉模型。这些模型单独轻量级，当为每个模型分配自己的 GPU 资源时，它们就会变得昂贵。

我们建议使用高密度 CPU 服务（使用 TorchServe 或 Triton 在 CPU 上为每个节点打包多个模型），由 Karpenter 管理节点生命周期，Keda 根据观察到的负载进行扩展。

这种模式自然延伸到 RAG 架构中：嵌入生成、文档分块和检索 OpenSearch 都在 CPU 节点上经济实惠地运行，仅在最后生成步骤中将结果提供给 GPU-hosted LLM。CPU 群负责处理音量；GPU 负责处理复杂性。

对于受监管的行业（金融服务、医疗保健、政府）来说，这种模式尤其引人注目：数百个专门的模型在CPU上运行在VPC内，具有完整的审计跟踪和永远不会离开账户的数据。自我管理推理的合规性要求自然符合低于 8B 模型的 CPU 成本优势。

## 优化最佳实践

### 量化

在 CPU 上以完整的 BF16 运行一个 7B 模型是不切实际的；在第 4 季度量化时运行它既可行又具有成本效益。了解量化为何对 CPU 有帮助，是做出正确基础设施决策的关键。

为什么量化对 CPU 推断很重要。CPU 推断受内存带宽限制，而不是计算限制。在解码阶段（一次生成一个代币），每个生成的代币都会从 RAM 中读取模型的全部权重。CPU 大部分时间都在等待数据从内存中到达，而不是进行数学运算。BF16的7B机型大约为14GB；在Q4_K_M，它缩小到大约4GB。由于瓶颈是将字节从RAM转移到CPU内核，因此小3.5倍的模型的读取速度提高了3.5倍，这几乎直接意味着令牌生成速度提高了3.5倍。这就是为什么量化是最有影响力的 CPU 推理优化的原因，也是为什么具有更多内存通道的新一代 CPU 即使在相同的时钟速度下也能产生更快的推理的原因。

我们建议使用架构优化的后端（ARM NEON/SVE2 适用于 arm64，对于 x86）构建推理引擎，AVX-512/AMX 将线程数设置为 vCPU 计数，然后选择 Q4_K_M 或 Q8_0 量化格式。

| 量化 | 质量影响 | 吞吐量与 BF16 对比 | 使用场景 |
| --- | --- | --- | --- |
| Q4_K_M | 低（1-3% 的困惑度增量，视型号而定） | 速度提高大约 4-5 倍 | SLM 的生产默认值 |
| Q8_0 | 可以忽略不计 | 速度提高了大约 2 倍 | Quality-sensitive 任务 |
| Q5_K_M | 非常低 | 速度提高了大约 3.5 倍 | 质量和速度的平衡 |
| BF16 | 无 | 1x（基准） | 对于 7B 以上的机型，请避免在 CPU 上使用 |

对于低于2b的机型，CPU在性价比上胜过GPU。这些模型足够小，GPU 加速带来的好处微乎其微，而每小时的成本要高得多。如果您的工作负载可以使用低于 2B 的模型，则建议使用默认配置 CPU。

Architecture-specific 优化：在 arm64 上，当前一代 Graviton 实例支持 SVE2。使用适合目标的`-march`标志来构建您的推理引擎。在 x86 上，AMD EPYC 实例支持 AVX-512，英特尔至强实例添加了 AMX 以实现矩阵加速。由于推理受内存带宽的限制，因此即使在相同的时钟速度下，具有更多 DDR5 内存通道的新一代 CPU 也能产生更快的推理。选择实例类型时，应优先考虑内存带宽而不是核心数量。

上下文窗口大小：对于分类和路由工作负载，输入通常低于 200 个标记，输出为 2-3 个标记。设置较小的上下文窗口（例如 512 个令牌）而不是默认的 2048，可降低 KV 缓存内存使用量并改善每个请求的延迟。只有当您的输入确实很长时，才会增加上下文窗口。

Flash 注意：如果您的推理引擎支持 Flash Attention，请启用。Flash Attention 通过避免实现完整的注意力矩阵来减少注意力计算的内存使用量。在CPU上，其优势要小于GPU，但对于较长的输入，它仍然有帮助。

> **Tip**
>
> Q4_K_M 质量下降因模型和任务而异。在部署到生产环境之前，请务必对自己的数据集进行评估。

## Bin-packing 适合密集食用

对于经典机器学习和嵌入模型（通常每个模型 <500MB），目标是在稳定的尾部延迟下每个节点的最大 pod 密度。有两个因素决定你能否实现这一目标：准确的资源请求和受控的线程。

`requests`根据实际负载下观察到的 p50-p90 使用情况。使用负载测试中的 Goldilocks、VPA 推荐或 Prometheus 直方图。两个方向的默认值几乎总是错误的。

机器学习库（PyTorch、ONNX Runtime、MKL、OpenBLAS）在节点上生成尽可能多的线程，而不是分配给 pod 的 CPU。在拥有 20 个 pod 的密集节点上，每个 Pod 都会尝试生成 32 个线程。该节点在上下文切换和 p99 延迟峰值时大放异彩。明确修复这个问题：

```
env:
  - name: OMP_NUM_THREADS
    value: "2"          # match your cpu request (2000m = 2 threads)
  - name: MKL_NUM_THREADS
    value: "2"
  - name: OPENBLAS_NUM_THREADS
    value: "2"
  - name: INTRA_OP_NUM_THREADS    # PyTorch / ONNX Runtime
    value: "2"
  - name: NUM_INTER_THREADS
    value: "1"          # keep inter-op parallelism minimal
```

将每个值设置为等于或低于您的 CPU 请求。对于拥有 4 个以上内核的 pod，基准测试从 2-4 个线程开始。由于缓存效率高，许多小型模型在使用更少的线程时性能更好。如果您将 HPA 与许多薄 pod 一起使用，则每个 Pod 有 1-2 个线程几乎总是会获胜。

## 日程安排和成本优化

两种做法相结合，可以显著降低 CPU 推断成本：采用 Karpenter 整合的 Spot 实例和多架构容器映像。

Karpenter 的整合非常适合 CPU 推理，因为队列或负载均衡器后面的无状态推理 pod 可以优雅地容忍中断。将整合配置为对未充分利用的节点采取行动，预算限制并发中断（例如，一次只能有 20% 的节

点），以避免在缩小规模期间容量下降。Karpenter 的nodePool规范允许你在单个池中混合 Spot 和 On-Demand 容量，将 Spot 作为首选选项和 On-Demand 备用选项。

构建多架构映像（arm64和amd64）可以进一步解锁这一点。由于这两种架构都可用，Karpenter 可以根据实时价格和可用性从全系列实例系列（Graviton、AMD、Intel）中进行选择。这对于竞价型工作负载特别有价值，在这些工作负载中，跨实例类型和架构进行多样化可以降低中断频率。使用docker buildx或具有多平台版本的 CI 管道来生成涵盖两种架构的单个清单。

## 容器启动优化

当 Karpenter 配置新节点（向上扩展、Spot 替换）时，容器运行时需要在 pod 启动之前提取推理图像。对于多 GB 的推理图像，这可能会延长 Pod 启动时间 30-60 秒。

我们建议使用 Bottlerocket 作为推理工作负载的节点操作系统，并在并行模式下与 SOCI 快照器结合使用。 pull/unpack SOCI 用基于并行区块的下载取代了默认的顺序图层提取，从而大大缩短了大图像的图像提取时间。无需更改您的容器镜像。

有关详细的配置指南，请参阅本指南的 “性能” 部分，其中包括 SOCI 配置、EBS 快照预提取和容器运行时缓存策略。

## 可观测性

如果模型层没有可观测性，你就是在盲目缩放。我们建议公开每项推理服务的 Prometheus 指标，并使用它们来推动 KEDA 扩展和操作仪表板。

大多数推理服务器（llama.cpp、vLLM、Triton 等 TorchServe）都会在端点上公开 Prometheus-compatible 指标。/metrics指标名称因服务器而异，但概念相同。

仪器的关键指标：

| 指标类别 | 说明 | 警报阈值 |
| --- | --- | --- |
| 请求处理/正在处理中 | 服务器当前正在处理的请求数。 | 用于缩放（参见下面的自动缩放部分） |
| 请求已排队/已延迟 | 等待处理时段的请求数。 | 缩放触发器。任何持续的队列都意味着延迟即将降低。 |

| 指标类别 | 说明 | 警报阈值 |
| --- | --- | --- |
| 代币吞吐量 | 每秒生成的代币。 | 如果负载下吞吐量降至基准的 50% 以下，则发出警报 |
| 请求延迟 | End-to-end 延迟直方图（提示处理+令牌生成）。 | 在 p95 超出你的 SLO 时发出警报 |
| KV 缓存利用率 | 键值缓存的填满程度（0.0 到 1.0）。接近 1.0 意味着服务器将开始拒绝请求或排队请求。 | 85% 以上时发出警报 |
| 容器内存 | 每个 pod 的 RSS 内存 (`container_memory_working_set_bytes`)。 | 在限制的 85% 时发出警报 |

## 自动扩展：根据队列深度进行扩展，而不是 CPU 利用率

CPU 利用率是一个饱和度指标。延迟已经降低后，它会达到峰值。当基于利用率的自动缩放做出反应时，用户已经在等待。

队列深度（请求 deferred/waiting）是一个领先指标。它会在延迟下降之前上升，因为当所有处理插槽都处于繁忙状态时，请求就会开始排队。扩展队列深度意味着在现有副本仍能正常响应的情况下配置新的副本。

KEDA 支持使用将多个指标合并到单个缩放公式中`scalingModifiers`（需要 KEDA 2.12+）。推理工作负载的推荐模式是将正在运行的请求与排队的请求相结合，从而对队列指标进行大量加权：

```
advanced:
  scalingModifiers:
    formula: "running + (waiting * 10)"
    target: "25"
    activationTarget: "5"
```

该公式`running + (waiting * 10)`意味着，即使有 3 个排队的请求也会将合并后的指标推至 55，远高于 25 的目标。在延迟下降之前开始扩展。of `activationTarget` 5 可防止噪声触发不必要的从零缩放的事件。

# 评估 CPU-First 工作负载的模型质量

在 CPU 上部署量化 SLM 是一个成本和延迟的决定。只有当模型仍然能为你的工作负载生成正确、有用的输出时，这才有意义。

较小的模型或量化可以降低计算成本，但会降低质量。影响各不相同。在 CPU 上运行良好的工作负载（分类、提取、路由、汇总、嵌入）通常通过适当的量化和提示在 3 个B-7B 范围内保持良好的质量。

## 要评估什么

不同的工作负载会以不同的方式降级：

| 工作负载 | 什么可能会降解 | 要衡量什么 |
| --- | --- | --- |
| 意图或门票分类 | 输入模棱两可时出错 | 精度，每级 F1 |
| 结构化提取 (JSON) | 缺少字段或架构错误 | 精确匹配，架构有效性 |
| RAG 答案 | 出现幻觉或忽视背景 | 忠诚，答案的相关性 |
| 总结 | 事实缺失或报道不佳 | ROUGE-L，bertScore，人工评论 |
| 代理路由 | 选择了错误的工具 | 刀具精度 |
| 嵌入 | 检索质量较差 | 回想一下 @K，NDCG |

## 实用的评估工作流程

我们建议在生产之前创建质量检查，类似于在选择实例类型之前运行负载测试的方式。该工作流程分为四个阶段：

1. 构建评估数据集 — 根据您的实际工作负载构建一个小型评估数据集（100-300 个带标签的示例）。避免使用像 MMLU 这样的通用基准来衡量一般推理，而不是你的实际任务。
2. 建立基准-根据可信模型运行数据集（例如，你知道的大型 LLM 会产生正确的结果）。
3. 测试 CPU 模型 — 在量化的 SLM 上运行相同的数据集并进行比较。
4. 评估-在测试之前定义质量阈值，例如，“SLM 准确度在基线的 5 个百分点以内”。正确的阈值取决于任务：与自动决策的系统相比，由人类审查的分类器可以容忍更多的错误。

### 如何恢复质量

如果模型表现不佳，请按努力顺序尝试以下方法：

- 在@@ 提示中添加少量示例：零成本，即时。在提示中包含 3-5 个带标签的示例通常可以缩小分类和提取任务的差距。
- 使用更高质量的量化格式：从第 4 季度移动到第 8 季度通常会恢复大部分丢失的质量，代价是内存增加大约 2 倍，吞吐量会降低。
- 使用混合路由：让 SLM 处理简单的案例，并将困难的输入发送到更大的模型。这是一项架构变更，但对于大多数流量，可以保持较低的 CPU 成本。
- Fine-tune 您所在域名的模型：最昂贵的选择，但最有效。LoRa Land（ar Xiv:2405.00732）的研究发现，经过微调的7B模型在测试的大多数特定领域任务中表现优 GPT-4 于其他领域。

## 参考

- 亚马逊 EKS 文档
- AWS Graviton 开发者指南
- Karpenter 文档
- KEDA 文档
- llama.cpp
- vLLM 文档
- 奥拉玛
- 射线服务文档
- 亚马逊 OpenSearch 服务
- AWS Spot 最佳实践
- LoRa Land：跑赢大盘的 Fine-Tuned Open-Source 法学硕士 GPT-4
- 模型上下文协议 (MCP) 规范
- Agentic AI 的小型语言模型 (NVIDIA)
- SOCI 快照器
- Bottlerocket 操作系统
- EKS 上的 AI：容器启动优化

# Networking

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

## 对于 Inter-Node 通信量大的应用程序，可以考虑使用更高的网络带宽或弹性结构适配器

对于节点间通信需求较高的 Amazon EKS 上的分布式训练工作负载，可以考虑选择网络带宽更高的实例或弹性结构适配器 (EFA)。网络性能不足会阻碍数据传输，从而减慢分布式多 GPU 训练等机器学习任务的速度。请注意，推理工作负载的节点间通信通常不高。

示例

例如，使用 Karpenter：

```
apiVersion: v1
kind: Pod
metadata:
  name: ml-workload
spec:
  nodeSelector:
    karpenter.k8s.aws/instance-network-bandwidth: "100000"  # 100 Gbps in Mbps
    node.kubernetes.io/instance-type: p5.48xlarge  # EFA-enabled instance
  containers:
  - name: training-job
    image: `763104351884.dkr.ecr.us-west-2.amazonaws.com/pytorch-inference:2.6.0-gpu-
py312-cu124-ubuntu22.04-ec2-v1.6`
    resources:
      limits:
        vpc.amazonaws.com/efa: 1  # Requires EFA device plugin
```

确保在您的容器映像中安装 MPI 和 NCCL 等工具，以便利用 EFA 进行训练作业。

## 规划大型 GPU 实例的 IP 地址消耗

默认情况下，Amazon VPC CNI 插件会预先分配 IP 地址以确保可以快速调度 pod，同时保留一个完整的备用 ENI 并填充 IP。在大型实例中，即使只有几个 pod 在运行，这也可能导致每个节点保留数十个 IP。

这种不匹配在每个节点 Pod 密度较低的训练和推理工作负载中很常见。在集群规模下，尤其是在启动许多 GPU 节点且每个 Pod 很少 GPU 节点的自动缩放事件中，即使实际 IP 利用率很低，这也可能导致子网 IP 耗尽。

要缓解这种情况，请调整WARM_IP_TARGETMINIMUM_IP_TARGET、和WARM_ENI_TARGET变量以匹配您的实际吊舱密度。有关更多信息，请访问 VPC CNI 的 ENI 和 IP 目标设置。

有关优化 IP 消耗的完整指南，请参阅优化 IP 地址利用率。

# 安全性

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

## 安全性与合规性

### 考虑使用 S3 和 KMS 进行符合加密标准的存储

除非您另行指定，否则SSE-S3默认情况下，所有 S3 存储桶都使用对静态对象进行加密。但是，您可以选择将存储桶配置为使用服务器端加密，改用 AWS Key Management Service (AWS KMS) 密钥 (SSE-KMS)。AWS KMS 中的安全控制措施可以帮助您满足与加密相关的合规性要求。您可以利用这些 KMS 密钥来保护在 Amazon S3 存储桶中的数据。当您对 S3 存储桶使用 SSE-KMS 加密时，AWS KMS 密钥必须与存储桶位于同一区域。

将您的通用存储桶配置为使用 S3 存储桶密钥 SSE-KMS，通过减少从 Amazon S3 到 AWS KMS 的请求流量，将您的 AWS KMS 请求成本降低多达 99%。S3 存储桶密钥始终为目录存储桶启用GET和PUT操作，并且无法禁用。

请注意，Amazon S3 Express One Zon e 使用一种名为 S3 目录存储桶的特定类型的存储桶。目录存储桶专门用于 S3 Express One Zone 存储类别，可实现高性能、低延迟的访问。要在 S3 目录存储桶上配置默认存储桶加密，请使用 AWS CLI 并指定 KMS 密钥 ID 或 ARN，而不是别名，如下例所示：

Example

```
aws s3api put-bucket-encryption --bucket my-directory-bucket --server-side-encryption-
configuration \
   '{"Rules": [{"ApplyServerSideEncryptionByDefault": {"SSEAlgorithm": "aws:kms",
 "KMSMasterKeyID": "1234abcd-12ab-34cd-56ef-1234567890ab"}}]}'
```

确保您的 EKS 容器的 IAM 角色具有 KMS 权限（例如`kms:Decrypt`），可以访问加密对象。在暂存环境中对此进行测试，方法是将示例模型上传到存储桶，将其安装到 pod 中（例如，通过 Mountpoint S3 CSI 驱动程序），并验证 pod 能否毫无错误地读取加密数据。通过 AWS 审核日志 CloudTrail ，以确认是否符合加密要求。有关设置详细信息和密钥管理，请参阅 KMS 文档。

# 仓储服务

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

## 数据管理和存储

### 使用 CSI 驱动程序将 AI 模型部署到 Pod

AI/ML 工作负载通常需要访问大型模型工件（例如经过训练的权重、配置），而 pod 需要一种可靠、可扩展的方式来访问这些工件，而无需将其嵌入容器镜像中，这可能会增加图像大小和容器注册表提取时间。为了减少管理卷挂载的运营开销，我们建议使用相应的 CSI 驱动程序将亚马逊存储服务（例如 S3、Lustre 的 fsX、OpenZFS 的 FSx、EFS 的 EFS）安装为永久卷 (PV)，将 AI 模型部署到容器中。有关实现的详细信息，请参阅本节的后续主题。

### 在 EKS 上优化 ML 模型缓存的存储

利用最佳存储解决方案对于最大限度地缩短 pod 和应用程序启动延迟、减少内存使用量、获得所需的性能水平以加速工作负载以及确保机器学习工作负载的可扩展性至关重要。机器学习工作负载通常依赖模型文件（权重），模型文件可能很大，需要在 Pod 或节点之间共享对数据的访问权限。选择最佳存储解决方案取决于您的工作负载的特征，例如单节点效率、多节点访问、延迟要求、成本限制以及数据集成要求（例如与 Amazon S3 数据存储库的集成）。我们建议根据您的工作负载对不同的存储解决方案进行基准测试，以了解哪一种解决方案符合您的要求，并且我们提供了以下选项来帮助您根据工作负载要求进行评估。

EKS CSI 驱动程序支持以下 AWS 存储服务，每种服务都有自己的 CSI 驱动程序，并且在 AI 和 ML 工作流程方面各有优势：

- 适用于 Amazon S3 的 Mountpoint
- Amazon FSx for Lustre
- Amazon FSx for OpenZFS

- Amazon EFS
- Amazon EBS

AWS 存储服务的选择取决于您的部署架构、规模、性能要求和成本策略。需要在您的 EKS 集群上安装存储 CSI 驱动程序，这允许 CSI 驱动程序在 Pod 生命周期之外创建和管理永久卷 (PV)。使用 CSI 驱动程序，您可以将支持的 AWS 存储服务创建为 EKS 集群资源的 PV 定义。然后，Pod 可以通过为 PV 创建永久卷声明 (PVC) 来访问这些存储卷作为其数据卷。根据 AWS 存储服务和您的部署方案，可以将单个 PVC（及其关联的 PV）连接到多个 Pod 以处理一个工作负载。例如，对于 ML 训练，共享的训练数据存储在 PV 上并由多个 Pod 访问；为了进行实时在线推理，LLM 模型缓存在一个 PV 上并由多个 Pod 访问。下面提供了 AWS 存储服务的示例 PV 和 PVC YAML 文件，以帮助您入门。

监控性能不佳的磁盘性能会延迟容器映像读取、增加 pod 启动延迟，并降低推理或训练吞吐量。使用 Amazon CloudWatch 监控您的 AWS 存储服务的性能指标。当您发现性能瓶颈时，请修改存储配置参数以优化性能。

场景：多个 GPU 实例工作负载

适用于 Lustre 的 Amazon FSx：如果您有多个 EC2 GPU 计算实例环境，其中包含延迟敏感和高带宽吞吐量的动态工作负载（例如分布式训练和模型服务），并且需要原生 Amazon S3 数据存储库集成，我们建议使用 Amazon FSx for Lustre。FSx for Lustre 提供了一个完全托管的高性能并行文件系统，专为高性能计算 (HPC)、Machine Learning 等计算密集型工作负载而设计。

您可以安装 FSx for Lustre CSI 驱动程序，将 FSx 文件系统作为永久卷 (PV) 挂载到 EKS 上，然后将 FSx for Lustre 文件系统部署为独立的高性能缓存或 S3-linked 文件系统以充当 S3 数据的高性能缓存，从而为通过 GPU 计算实例访问数据 I/O 提供快速而高的吞吐量。FSx for Lustre 可以通过以下 Persistent-SSD 任一存储选项 Scratch-SSD 进行部署：

- Scratch-SSD 存储：建议用于临时或短期（小时）的工作负载，每个 Tib 预置的吞吐量容量是固定的。
- Persistent-SSD 存储：建议用于需要最高可用性的任务关键型、长时间运行的工作负载，例如 HPC 模拟、大数据分析或 Machine Learning 训练。使用 Persistent-SSD 存储，您可以配置所需的存储容量和吞吐容量（每个 TiB）。

性能注意事项：

- 用于管理 FSx for Lustre 文件系统的管理 pod：配置一个安装了 lustre 客户端并安装了 FSx 文件系统的 “管理” Pod。这将使接入点能够对 FSx 文件系统进行微调，也可以在启动 GPU 计算实例之前使用 ML 训练数据或 LLM 模型预热 FSx 文件系统的情况下。如果您的架构使用 Spot-based

Amazon EC2 GPU/compute 实例，则这一点尤其重要，您可以在其中使用管理 Pod 将所需数据 “预热” 或 “预加载” 到 FSx 文件系统中，以便在运行基于竞价的 Amazon EC2 实例时可以处理数据。

- 弹性结构适配器 (EFA) Persistent-SSD ：存储部署类型支持弹性结构适配器 (EFA)，其中使用 EFA 非常适合高性能和基于吞吐量的工作负载。 GPU-based 请注意，FSx for Lustre 支持 NVIDIA GPUDirect 存储 (GDS)，其中 GDS 是一种在本地或远程存储与 GPU 内存之间创建直接数据路径的技术，以实现更快的数据访问。
- 压缩：如果您的文件类型可以压缩，请在文件系统上启用数据压缩。这有助于提高性能，因为数据压缩可以减少在 FSx for Lustre 文件服务器和存储之间传输的数据量。
- Lustre 文件系统条带化配置：
  - 数据条带化：允许 FSx for Luster 在 Lustre 文件系统中的多个对象存储目标 (OST) 上分发文件数据，从而最大限度地提高并行访问和吞吐量，尤其是在大规模机器学习训练作业中。
  - 独立文件系统条带化：默认情况下，通过 FSx for Lustre 的渐进式文件布局 (PFL) 功能为您创建一个 4 组件 Lustre 条带化配置。在大多数情况下，你不需要更新默认的 PFL Lustre 条纹。 count/size如果需要调整 Lustre 数据条带，则可以通过参考 FS x for Lustre 文件系统的条带化参数来手动调整 Lustre 条带。
  - S3-Linked 文件系统：使用原生 Amazon S3 集成（数据存储库关联或 DRA）导入到 FSx 文件系统的文件不使用默认 PFL 布局，而是使用文件系统参数中的布局。`ImportedFileChunkSize` S3-imported 大于的文件`ImportedFileChunkSize`将存储在多个 OST 上，条带计数基于`ImportedFileChunkSize`定义的值（默认为 1GiB）。如果您有较大的文件，我们建议将此参数调整为更高的值。
  - 放置：将 FSx for Lustre 文件系统部署在与计算或 GPU 节点相同的可用区中，以实现最低延迟的数据访问，避免跨可用区访问模式。如果您有多个 GPU 节点位于不同的可用区，那么我们建议您在每个可用区部署一个 FSx 文件系统，以实现低延迟的数据访问。

示例

FSx for Lustre 文件系统的永久卷 (PV) 定义，使用静态配置（其中已配置 FSx 实例）。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fsx-pv
spec:
  capacity:
    storage: 1200Gi
  volumeMode: Filesystem
  accessModes:
```

```
    - ReadWriteMany
  mountOptions:
    - flock
  persistentVolumeReclaimPolicy: Recycle
  csi:
    driver: fsx.csi.aws.com
    volumeHandle: [FileSystemId of FSx instance]
    volumeAttributes:
      dnsname: [DNSName of FSx instance]
      mountname: [MountName of FSx instance]
```

示例

PV 的永久容量声明定义为`fsx-pv`：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fsx-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 1200Gi
  volumeName: fsx-pv
```

示例

将 Pod 配置为使用永久卷声明`fsx-claim`：

```
apiVersion: v1
kind: Pod
metadata:
  name: fsx-app
spec:
  containers:
  - name: app
    image: amazonlinux:2023
    command: ["/bin/sh"]
    volumeMounts:
    - name: persistent-storage
```

```
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: fsx-claim
```

有关完整示例，请参阅中的 FSx for Lustre 驱动程序示例。 GitHub使用亚马逊监控亚马逊 FSx 的 Lustre 绩效指标。 CloudWatch发现性能瓶颈后，请根据需要调整配置参数。

场景：单 GPU 实例工作负载

带有 CSI 驱动程序的 Amazon S3 的 Mountpoint：你可以使用 Moun tpoint for Amazon S3 CSI 驱动程序将 S3 存储桶作为卷挂载到你的 pod 中。此方法允许对哪些 Pod 可以访问特定的 S3 存储桶进行精细的访问控制。每个 Pod 都有自己的挂载点实例和本地缓存（5-10GB），从而隔离了 Pod 之间的模型加载和读取性能。此设置支持使用服务账户的 IAM 角色 (IRSA) 进行 pod 级身份验证，并支持针对不同模型或客户的独立模型版本控制。权衡是内存使用量和 API 流量增加，因为每个 pod 都会发出 S3 API 调用并维护自己的缓存。

示例使用 CSI 驱动程序的 Pod 部署 YAML 的部分示例：

```
# CSI driver dynamically mounts the S3 bucket for each pod

volumes:
  - name: s3-mount
    csi:
      driver: s3.csi.aws.com
      volumeAttributes:
        bucketName: your-s3-bucket-name
        mountOptions: "--allow-delete"  # Optional
        region: us-west-2

containers:
  - name: inference
    image: your-inference-image
    volumeMounts:
      - mountPath: /models
        name: s3-mount
volumeMounts:
  - name: model-cache
    mountPath: /models
volumes:
  - name: model-cache
```

```
    hostPath:
      path: /mnt/s3-model-cache
```

性能注意事项：

- 数据缓存：适用于 S3 的 Mountpoint 可以缓存内容以降低成本并提高重复读取同一文件的性能。有关缓存选项和参数，请参阅缓存配置。
- 对象部分大小：存储和访问大小超过 72GB 的文件时，请参阅配置 Mountpoint 性能，了解如何配置`--read-part-size`和`--write-part-size`命令行参数以满足您的数据配置文件和工作负载要求。
- Shared-cache专为大小不超过 1MB 的对象而设计。它不支持大型对象。使用本地缓存选项在 EKS 节点上缓存 NVMe 或 EBS 卷中的对象。
- API 请求费用：使用适用于 S3 的 Mountpoint 执行大量文件操作时，API 请求费用可能会成为存储成本的一部分。为了缓解这种情况，如果不需要强一致性，请务必启用元数据缓存，并将`metadata-ttl`时间设置为将 API 操作数量减少到 S3。

有关更多详细信息，请参阅亚马逊 EKS 官方文档中的 Mountpoint for Amazon S3 CSI 驱动程序。如果出现瓶颈，我们建议使用亚马逊指标监控 Amazon S3 的性能 CloudWatch 指标，并在需要时调整配置。

### 适用于 OpenZFS 的亚马逊 FSX 永久共享存储

对于涉及多个 EC2 GPU 计算实例的场景，这些实例的延迟敏感型工作负载需要高可用性、高性能、成本敏感性，以及针对不同应用程序部署多个 Pod，我们建议使用 Amazon FSX for OpenZFS。一些工作负载示例包括实时推理、强化学习和训练生成对抗网络。FSx for OpenZFS 对于需要高性能访问具有小文件且使用小 IO 数据访问模式的小文件的高性能目录结构的工作负载特别有用。此外，FSx for OpenZFS 还提供了独立于存储容量扩展性能的灵活性，在保持所需性能水平的同时，将存储大小与实际需求相匹配，从而帮助您实现最佳成本效益

OpenZFS 的本机 FSX CSI 驱动程序允许通过创建多个卷来为单个文件系统创建多个 PVC。通过在单个文件系统上进行整合的应用程序容器部署，这减少了管理开销，最大限度地提高了文件系统的吞吐量和 IOPS 的利用率。此外，它还包括企业功能，例如零副本快照、零副本克隆以及可以通过 CSI 驱动程序动态配置的用户和组配额。

适用于 OpenZFS 的 FSX 在创建时支持三种不同的部署类型：

- Single-AZ：成本最低的选项，延迟时间为亚毫秒，但在文件系统或可用区级别不提供高可用性。建议用于开发和测试工作负载或在应用程序层具有高可用性的工作负载。

- Single-AZ (HA)：在文件系统级别提供高可用性，延迟为亚毫秒。建议用于需要高可用性的最高性能工作负载。
- Multi-AZ：在文件系统级别以及跨可用区域提供高可用性。建议用于需要跨可用区提高可用性的高性能工作负载。

性能注意事项：

- 部署类型：如果不需要跨可用区的额外可用性，请考虑使用 Single-AZ (HA) 部署类型。这种部署类型可提供高达 100% 的写入吞吐量，保持亚毫秒级的延迟，而且 Gen2 文件系统还有一个额外的 NVMe 缓存，用于存储多达 TB 的频繁访问数据。 Multi-AZ 文件系统以更高的延迟为写入提供高达 75% 的吞吐量，以适应跨可用区的流量。
- 吞吐量和 IOPS：为文件系统配置的吞吐量和 IOPS 都可以在部署后向上或向下扩展。您最多 10GB/s 可以预配置磁盘吞吐量，从而提供最多 21GB/s 缓存的数据访问权限。IOPS 可以从磁盘扩展到 400,000，缓存可以提供超过 100 万的 IOPS。请注意，由于不存在高可用性， Single-AZ 文件系统的吞吐量扩展确实会导致文件系统短暂中断。可以无中断地扩展 Single-AZ (HA) 或 Multi-AZ 文件系统的吞吐量。固态硬盘 IOPS 可以每六小时扩展一次。
- 存储类别：适用于 OpenZFS 的 FSX 既支持固态硬盘存储类别，也支持存储类别。Intelligent-Tiering对于 AI/ML 工作负载，建议使用固态硬盘存储类别，为工作负载提供稳定的性能，使CPU s/GPU 尽可能保持繁忙。
- 压缩：如果您的工作负载可以压缩，请启用 LZ4 压缩算法。这减少了每个文件在缓存中消耗的数据量，从而可以直接从缓存中提供更多数据，因为网络吞吐量和 IOPS 可以减少 SSD 磁盘上的负载。
- 记录大小：保留默认的 128KiB 记录大小将使大多数 AI/ML 工作负载受益。只有当数据集由大文件（大于 10GiB）组成，且应用程序对小块的访问量始终低于 128KiB 时，才应降低此值。

创建文件系统后，服务会自动创建关联的根卷。最佳做法是将数据存储在文件系统上根卷的子卷中。使用 FSX for OpenZFS CSI 驱动程序，您可以创建关联的永久卷声明，以动态创建子卷。

示例：

适用于 OpenZFS 卷的 FSX 的存储类别 (SC) 定义，用于在现有文件系统上创建根卷 ($ROOT_VOL_ID) 的子卷，然后使用 NFS v4.2 协议将该卷导出到 VPC CIDR ($VPC_CIDR)。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fsxz-vol-sc
```

```
provisioner: fsx.openzfs.csi.aws.com
parameters:
  ResourceType: "volume"
  ParentVolumeId: '"$ROOT_VOL_ID"'
  CopyTagsToSnapshots: 'false'
  DataCompressionType: '"LZ4"'
  NfsExports: '[{"ClientConfigurations": [{"Clients": "$VPC_CIDR", "Options":
 ["rw","crossmnt","no_root_squash"]}]}]'
  ReadOnly: 'false'
  RecordSizeKiB: '128'
  Tags: '[{"Key": "Name", "Value": "AI-ML"}]'
  OptionsOnDeletion: '["DELETE_CHILD_VOLUMES_AND_SNAPSHOTS"]'
reclaimPolicy: Delete
allowVolumeExpansion: false
mountOptions:
  - nfsvers=4.2
  - rsize=1048576
  - wsize=1048576
  - timeo=600
  - nconnect=16
  - async
```

针对上面创建的 fsxz-vol-sc 动态创建的永久卷声明 (PVC)。注意，分配的存储容量为 1Gi，如 CSI 驱动程序常见问题解答中所述，这是 FSX 的 OpenZFS 卷所必需的。将为该卷提供使用此配置为文件系统配置的全部容量。如果需要限制卷容量，则可以使用用户或组配额进行限制。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-vol-pvc
  namespace: example
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: fsxz-vol-sc
  resources:
    requests:
      storage: 1Gi
```

使用 dynamic-vol-pvc 的永久卷声明 (PVC) 将 pod 配置为挂载卷：

```
kind: Pod
```

```
apiVersion: v1
metadata:
  name: fsx-app
  namespace: example
spec:
  volumes:
    - name: dynamic-vol-pv
      persistentVolumeClaim:
        claimName: dynamic-vol-pvc
  containers:
    - name: app
      image: amazonlinux:2023
      command: ["/bin/sh"]
      volumeMounts:
        - mountPath: "/mnt/fsxz"
          name: dynamic-vol-pv
```

## 适用于共享模型缓存的 Amazon EFS

如果您有多个 EC2 GPU 计算实例环境，并且动态工作负载需要跨多个节点和可用区共享模型访问权限（例如，使用 Karpenter 进行实时在线推理），且性能和可扩展性需求适中，我们建议通过 EFS CSI 驱动程序将 Amazon Elastic File System (EFS) 文件系统用作永久卷。Amazon EFS 是一个完全托管、高度可用且可扩展的基于云的 NFS 文件系统，它使 EC2 实例和容器具有共享文件存储，性能稳定，无需预先配置存储。使用 EFS 作为模型卷，并通过在 EKS 集群上定义永久卷将该卷作为共享文件系统挂载。由 EFS 文件系统支持的每个永久卷声明 (PVC) 都是作为 EFS Access-point 文件系统的 EFS 创建的。EFS 允许多个节点和 pod 访问相同的模型文件，无需将数据同步到每个节点的文件系统。安装 EFS CSI 驱动程序，将 EFS 与 EKS 集成。

您可以使用以下吞吐量模式部署 Amazon EFS 文件系统：

- 突增吞吐量：根据文件系统大小扩展吞吐量，适用于偶尔出现突发的不同工作负载。
- 预配置吞吐量：专用吞吐量，非常适合在限制范围内具有可预测性能需求的一致机器学习训练作业。
- 弹性吞吐量（建议用于机器学习）：根据工作负载自动扩展，针对不同的机器学习工作负载实现成本效益。

要查看性能规格，请参阅 Amazon EFS 性能规范。

性能注意事项：

- 使用弹性吞吐量处理不同的工作负载。

- 对活跃的 ML 工作负载使用标准存储类。

有关在 EKS 集群和 Pod 中使用 Amazon EFS 文件系统作为永久卷的完整示例，请参阅中的 EFS CSI 驱动程序示例 GitHub。使用亚马逊监控亚马逊 EFS 性能指标 CloudWatch。发现性能瓶颈后，请根据需要调整配置参数。

## 使用 S3 Express One Zone 进行 Latency-Sensitive面向对象的工作流程

对于 Amazon EKS AI/ML 上对延迟敏感的工作负载，例如大规模模型训练、推理或高性能分析，我们建议使用 S3 Express One Zon e 进行高性能模型存储和检索。S3 Express One Zone 提供了一个分层命名空间，比如文件系统，你只需在其中上传到目录存储桶，适合 “将所有内容放进去”，同时保持高速度。如果您习惯了面向对象的工作流程，则此功能特别有用。或者，如果你更习惯文件系统（例如 POSIX-compliant），你可能更喜欢亚马逊 FSX for Lustre 或 OpenZFS。Amazon S3 Express One Zone 使用目录存储桶将数据存储在单个可用区 (AZ) 中，其延迟低于在多个可用区之间分配数据的标准 S3 存储桶。为了获得最佳结果，请确保将 EKS 计算与您的 Express One Zone 存储桶放在同一个可用区中。要详细了解 S3 Express One Zone 的区别，请参阅目录存储桶的差异。

要使用文件系统语义访问 S3 Express One Zone，我们建议使用 Mountpoint S3 CSI 驱动程序，该驱动程序将 S3 存储桶（包括 Express One Zone）挂载为本地文件系统。这将文件操作（例如打开、读取、写入）转换为 S3 API 调用，从而提供针对来自多个客户端的读取密集型工作负载和顺序写入新对象而优化的高吞吐量访问权限。有关支持的操作和限制（例如，不完全符合 POSIX，但 Express One Zone 支持附加和重命名）的详细信息，请参阅 Moun tpoint 语义文档。

### 性能优势

- 数据访问速度比 S3 标准版快 10 倍，延迟保持在个位数毫秒，请求成本最多可降低 80%。
- 可扩展到每个目录存储桶每秒处理数十万至数百万个请求，从而避免了在极端负载期间（例如，来自具有数万到数十万个 GPUs/CPUs 饱和网络的集群）在标准 S3 中出现的限流或停电。
- 使用基于会话的身份验证机制。进行一次身份验证即可获得会话令牌，然后高速重复执行操作，而不会产生每个请求的身份验证开销。这针对频繁的检查点操作或数据加载等工作负载进行了优化。

### 推荐用例

- 缓存：在 S3 Express One Zone 中使用 Mountpoint S3 CSI 驱动程序的首要用例之一是缓存。第一个实例从 S3 标准（通用）读取数据，将其缓存到延迟较低的 Express One Zone 中。其他客户端的后续读取可以更快地访问缓存的数据，这非常适合多个 EKS 节点读取相同数据（例如，共享训练数据集）的多节点场景。这可以将重复访问的性能提高多达 7 倍，并降低计算成本。对于需要完全

符合 POSIX 标准的工作负载（例如文件锁定和就地修改），可以考虑将 Amazon FSX for Lustre 或 OpenZFS 作为替代方案。

- Large-Scale AI/ML 训练和推理：非常适合具有成百上千个计算节点（例如 EKS 集群中的 GPU）的工作负载，在这些工作负载中，通用 S3 限制可能会导致延迟，浪费昂贵的计算资源。例如，法学硕士研究人员或运行每日模型的组织可以从快速、可靠的访问中 tests/checkpoints 受益，而不会破坏区域S3。对于规模较小的工作负载（例如 10 个节点），S3 标准或其他存储类别可能就足够了。
- 数据管道： Load/prepare 模型、存档训练数据或流检查点。如果您的团队更喜欢对象存储而不是传统文件系统（例如，由于熟悉 S3），请使用它来代替对 FSx for Lustre 等 POSIX-compliant 选项进行工程更改。

注意事项

- 弹性： Single-AZ 设计提供 99.999999999% 的耐久性（与标准 S3 相同，通过可用区内的冗余），但与多可用区类别（99.99% 的可用性）相比，可用性较低（设计为 99.95%，服务级别协议 99.9%）（可用性为 99.99%）。它对可用区故障的弹性较差。用于可重新创建或缓存的数据。考虑对关键工作负载进行多可用区复制或备份。
- API and Feature Support：支持 S3 API 的子集（例如，没有生命周期策略或复制）；可能需要对应用程序进行细微更改才能进行会话身份验证或对象处理。
- EKS 集成： Co-locate 您的 EKS 与目录存储桶 pods/nodes 位于同一个可用区，以最大限度地减少网络延迟。使用适用于 Amazon S3 的 Mountpoint 或 CSI 驱动程序进行访问。 Kubernetes-native
- 测试：在非生产环境中测试检索延迟，以验证性能提升。监控标准 S3 场景（例如 GPU 饱和度高）中的限制并进行比较。

S3 Express One Zone 存储类可在多个区域使用，并与 EKS 集成，适用于需要访问对象而无需等待存储的工作负载。要了解更多信息，请参阅 S3 Express One 区域入门。

# 可观测性

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

# 监控和可观测性

## GPU 指标详解

GPU 利用率指标显示 GPU 在示例窗口中是否运行了任何工作。该指标捕获 GPU 执行至少一条指令的时间百分比，但它并未显示 GPU 使用其硬件的效率。GPU 包含多个流式多处理器 (SM)，它们是执行指令的并行处理单元。100% 的利用率读数可能意味着 GPU 在其所有 SM 上运行了繁重的并行工作负载，也可能意味着在采样周期内，一条小指令激活了 GPU。要了解实际利用率，您需要检查硬件架构的多个级别的 GPU 指标。每个流式多处理器都由不同的内核类型构建，并且每层都暴露出不同的性能特征。 Top-level 指标（GPU 利用率、内存利用率、GPU 功率和 GPU 温度，可通过 nvidia-smi 查看）显示设备是否处于活动状态。更深入的指标（SM 利用率、SM 活动和张量内核使用率）揭示了 GPU 使用其资源的效率。

## 瞄准高 GPU 功耗使用率

未充分利用的 GPU 会浪费计算容量并增加成本，因为工作负载无法同时使用所有 GPU 组件。对于 Amazon EKS 上的 AI/ML 工作负载，请作为代理跟踪 GPU 功耗以识别实际的 GPU 活动。GPU 利用率报告 GPU 执行任何内核的时间百分比，但它不会显示流式多处理器、内存控制器和张量内核是否同时处于活动状态。功耗暴露了这种差距，因为与运行轻量级内核或在任务之间闲置的硬件相比，完全投入的硬件消耗的电量要多得多。将功耗与 GPU 的散热设计能力 (TDP) 进行比较，以发现利用率不足，然后调查您的工作负载是否受到 CPU 预处理 I/O、网络或批量大小效率低下的瓶颈。

在 Amazon EKS 上设置 Contain CloudWatch er Insights，以识别 GPU 功耗较低的 pod、节点或工作负载。该工具直接与 Amazon EKS 集成，允许您监控 GPU 功耗，并在用电量低于目标水平时调整 Pod 调度或实例类型。如果您需要高级可视化或自定义仪表板，请使用 DCGM-Exporter 带有 Prometheus 和 Grafana 的 NVIDIA 进行监控。 Kubernetes-native 两者都接近NVIDIA的关键指标，例如`nvidia_smi_power_draw`（GPU功耗）和`nvidia_smi_temperature_gpu`（GPU温度）。有关指标列表，https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-NVIDIA-GPU.htm.请浏览查找模式，例如在特定时间或特定工作中持续低功耗。这些趋势可帮助您确定在何处整合工作负载或调整资源分配。

Kubernetes 中的静态资源限制（例如 CPU、内存和 GPU 数量）通常会导致过度配置或利用不足，尤其是对于需求波动的推理等动态 AI/ML 工作负载。分析您的利用率趋势，将工作负载整合到更少的 GPU 上。在分配其他 GPU 之前，请确保每个 GPU 达到最大利用率。这种方法可以减少浪费并降低成本。有关优化调度和共享策略的详细指南，请参阅 EKS 计算和自动缩放最佳实践

# 可观察性和指标

## 为您的 AI/ML 工作负载使用监控和可观测性工具

现代 AI/ML 服务需要在基础架构、建模和应用程序逻辑之间进行协调。平台工程师管理基础设施和可观测性堆栈。他们收集、存储和可视化指标。 AI/ML 工程师定义特定于模型的指标，并在不同的负载和数据分布下监控性能。应用程序开发人员使用 API、路由请求并跟踪服务级别指标和用户互动。如果没有统一的可观测性实践，这些团队就会各自为政，错过有关系统运行状况和性能的关键信号。建立跨环境的共享可见性可确保所有利益相关者都能及早发现问题并保持可靠的服务。

针对 AI/ML 工作负载优化 Amazon EKS 集群带来了独特的监控挑战，尤其是在GPU内存管理方面。如果没有适当的监控，组织就会面临内存不足 (OOM) 错误、资源效率低下和不必要的成本。有效的监控可确保 EKS 客户获得更好的性能、弹性和更低的成本。使用结合三个监控层的整体方法。首先，使用 NVIDIA DCGM Exporter 监控精细的 GPU 指标，以跟踪 GPU 功耗使用情况、GPU 温度、SM 活动、SM 占用率和 XID 错误。其次，监控 Ray 和 vLLM 等推理服务框架，通过其原生指标获得分布式工作负载见解。第三，收集应用程序级见解，以跟踪特定于您的工作负载的自定义指标。这种分层方法使您可以从硬件利用率到应用程序性能进行全面了解。

### 工具和框架

有几种工具和框架为监控 AI/ML 工作负载提供了开箱即用的原生指标。这些内置指标消除了对自定义仪器的需求，并缩短了设置时间。这些指标侧重于性能方面，例如延迟、吞吐量和令牌生成，这些方面对于推理服务和基准测试至关重要。使用原生指标可以立即开始监控，而无需构建自定义收集管道。

- vLLM：适用于大型语言模型 (LLM) 的高吞吐量服务引擎，可提供请求延迟和内存使用量等原生指标。
- Ray：一种分布式计算框架，它为可扩展的 AI 工作负载发布指标，包括任务执行时间和资源利用率。
- Hugging Face 文本生成推理 (TGI)：用于部署和提供 LLM 的工具包，内置推理性能指标。
- NVIDIA genai-perf：一款命令行工具，用于对生成式 AI 模型进行基准测试，测量吞吐量、延迟和 LLM-specific 指标，例如在特定时间间隔内完成的请求。

### 可观测性方法

我们建议通过以下方式之一实现任何其他可观测性机制。

CloudWatch 容器见解如果您的组织更喜欢设置最少的 AWS-native 工具，我们建议使用CloudWatch 容器见解。它与 NVIDIA DCGM Exporter 集成以收集 GPU 指标，并提供预先构建的仪表板以快

速获得见解。通过在集群上安装CloudWatch 可观察性插件，Container Insights 可以部署和管理 NVIDIA DCGM Exporter 的生命周期，该导出器从 Nvidia 的驱动程序中收集 GPU 指标并将其公开。 CloudWatch

安装 Container Insights 后， CloudWatch 会自动检测环境中的 NVIDIA GPU 并收集关键的运行状况和性能指标。这些指标显示在精心策划的开箱即用仪表板上。您还可以 CloudWatch 使用 Unif ied A CloudWatch gent 将 Ray 和 vLLM 与发送其原生指标进行集成。这种统一的方法简化了 EKS 环境中的可观察性，让团队可以专注于性能调整和成本优化，而不是构建监控基础架构。

有关可用指标的完整列表，请参阅 Amazon EKS 和 Kubernetes 容器洞察指标。有关实施 GPU 监控的分步指南，请参阅使用 Amazon Contain CloudWatch er Insights 获取 NVIDIA GPU 工作负载的运营见解。有关优化推理延迟的实际示例，请参阅优化 AI 响应能力：Amazon Bedrock 延迟优化推理实用指南。

托管 Prometheus 和 Grafana 如果您的组织需要自定义仪表板和高级可视化功能，请部署 Prometheus 和 NVIDIA 和 Grafana 进行监控。 DCGM-Exporter Kubernetes-native Prometheus 从中抓取和存储 GPU 指标 DCGM-Exporter，而 Grafana 则提供灵活的可视化和警报功能。与 Contain CloudWatch er Insights 相比，这种方法可以让你更好地控制仪表板设计和指标保留率。

您可以通过集成 Ray 和 vLLM Ray 和 vLLM 等开源框架来扩展此监控堆栈，将其原生指标导出到 Promethe us。您还可以将 Grafana 连接到 AW X-Ray S 数据源，以可视化分布式跟踪并识别推理管道中的性能瓶颈。这种组合提供了从 GPU-level 指标到应用程序级请求流的端到端可见性。

有关部署此监控堆栈的分步指南，请参阅使用 AWS 托管开源服务在 Amazon EKS 上监控 GPU 工作负载。

### 考虑监控核心训练& Fine-Tuning 指标

监控核心训练指标，跟踪 Amazon EKS 集群的运行状况和性能以及在其上运行的机器学习工作负载。训练工作负载的监控要求与推理工作负载不同，因为它们运行时间长，消耗资源的方式不同，并且需要对模型融合和数据管道效率的可见性。以下指标可帮助您识别瓶颈、优化资源分配并确保训练作业成功完成。有关实施此监控方法的分步指南，请参阅在 Amazon EKS 上观察机器学习工作负载简介。

资源使用量指标

监控资源使用指标，以验证您的资源是否被正确消耗。这些指标可帮助您确定瓶颈和性能问题的根本原因。

- CPU、内存、网络、GPU 功率和 GPU 温度-监控这些指标，确保分配的资源满足工作负载需求并确定优化机会。跟踪 gpu_memory_usage_bytes 等指标，以识别内存消耗模式并检测峰值使用情况。

计算百分位数，例如第 95 个百分位数 (P95)，以了解训练期间的最高内存需求。此分析可帮助您优化模型和基础架构，以避免 OOM 错误并降低成本。

- SM 占用率、SM 活动、FpxX 活动 ——监控这些指标以了解 GPU 底层资源的使用情况。根据经验，SM Activity 的目标为 0.8。
- 节点和 Pod 资源利用率-跟踪节点和 Pod 级别的资源使用情况，以确定资源争用和潜在瓶颈。监控节点是否接近容量限制，这可能会延迟 Pod 调度并减慢训练作业。
- 资源利用率与请求和限制的比较-将实际资源使用量与配置的请求和限制进行比较，以确定您的集群能否处理当前工作负载并适应未来的工作负载。此比较揭示了您是否需要调整资源分配以避免 OOM 错误或资源浪费。
- 来自机器学习框架的内部指标-从机器学习框架（例如和）中捕获内部训练 TensorFlow 和融合指标 PyTorch。这些指标包括损失曲线、学习率、批处理时间和训练步骤持续时间。使用 TensorBoard 或类似的工具可视化这些指标，以跟踪模型收敛性并识别训练效率低下的情况。

### 模型性能指标

监控模型性能指标，以验证您的训练过程生成的模型是否符合准确性和业务要求。这些指标可帮助您确定何时停止训练、比较模型版本以及识别性能下降情况。

- 准确性、精度、召回率和 F1-score — 跟踪这些指标，以了解您的模型在验证数据上的表现如何。F1-score 在每个训练周期结束后计算验证集，以评估模型是否在改进以及何时达到可接受的性能水平。
- Business-Specific 指标和关键绩效指标-定义和跟踪可直接衡量计划业务价值的指标。 AI/ML 对于推荐系统，跟踪点击率或转化率等指标，以确保模型推动预期的业务成果。
- 一段时间内的性能-比较不同模型版本和训练运行的性能指标，以确定趋势并检测性能下降。与基准模型相比，跟踪较新的型号版本是保持还是提高了性能。这种历史比较可以帮助您决定是部署新模型还是调查训练问题。

### 数据质量和漂移指标

监控数据质量和偏差指标，确保您的训练数据保持一致和具有代表性。随着时间的推移，数据漂移会导致模型性能下降，而数据质量问题可能会导致模型无法收敛或产生不可靠的结果。

- 输入数据的统计特性-跟踪统计属性，例如平均值、标准差和输入特征随时间的分布，以检测数据漂移或异常。监控特征分布是否与基线训练数据相比有显著变化。例如，如果关键要素的均值变化超过两个标准差，请调查您的数据管道是否已更改或底层数据源是否发生了变化。

- 数据漂移检测和警报-实施自动化机制，在数据质量问题影响训练之前对其进行检测和提醒。使用诸如检验或卡方 Kolmogorov-Smirnov 检验之类的统计检验，将当前的数据分布与原始训练数据进行比较。在测试检测到明显偏差时设置警报，以便您可以使用更新的数据重新训练模型或调查数据管道问题。

### 延迟和吞吐量指标

监控延迟和吞吐量指标，以识别训练管道中的瓶颈并优化资源利用率。这些指标可帮助您了解训练期间花在何处以及将优化工作重点放在哪里。

- End-to-End 机器学习训练管道的延迟 — 测量数据流经整个训练管道的总时间，从数据摄取到模型更新。在整个训练运行中跟踪该指标，以确定管道变化是提高还是降低了性能。高延迟通常表示节点间的数据加载、预处理或网络通信存在瓶颈。
- 训练吞吐量和处理率 — 跟踪您的训练管道在单位时间内处理的数据量，以确保有效的资源利用率。监控指标，例如每秒处理的样本或每分钟完成的批次。相对于硬件容量而言，吞吐量较低意味着数据加载、预处理或模型计算效率低下，从而浪费 GPU 周期。
- 检查点保存和恢复延迟 — 监控将模型检查点保存到存储（S3、EFS、FSx），并在恢复作业或从故障中恢复时将其恢复到 GPU 或 CPU 内存所需的时间。缓慢的检查点操作会延长任务恢复时间并增加成本。跟踪检查点大小、保存持续时间、恢复持续时间和失败次数，以确定优化机会，例如压缩或更快的存储层。
- 数据加载和预处理时间-测量从存储中加载数据和应用预处理转换所花费的时间。将此时间与模型计算时间进行比较，以确定您的训练是数据绑定还是计算绑定。如果数据加载消耗的训练时间超过总训练时间的 20%，可以考虑通过缓存、预取或更快的存储来优化数据管道。

### 错误率和失败率

监控整个训练管道的错误率和失败率，以保持可靠性并防止浪费计算资源。未被发现的错误可能会导致训练作业静默失败，生成无效的模型，或者在你发现问题之前浪费数小时的 GPU 时间。

- 管道错误监控-跟踪机器学习管道所有阶段的错误，包括数据预处理、模型训练和检查点操作。记录错误类型、频率和受影响的组件，以快速识别问题。常见错误包括数据格式不匹配、预处理期间出现内存不足故障，以及存储限制导致的检查点保存失败。在错误率超过基线阈值时设置警报，以便在错误级联之前进行调查。
- 重复错误分析 — 识别和调查反复出现的错误模式，以防止 future 出现故障并提高管道可靠性。分析日志，以了解特定的数据样本、批量大小或训练配置是否始终导致故障。例如，如果某些输入数据类型触发了预处理错误，请在管道的早期添加验证检查或更新数据清理逻辑。跟踪平均故障间隔时间 (MTBF)，以衡量管道可靠性是否随着时间的推移而改善。 ”

### Kubernetes 和 EKS 的特定指标

监控 Kubernetes 和 EKS 指标，确保您的集群基础设施保持健康并能够支持您的训练工作负载。这些指标可帮助您在基础设施问题导致训练作业失败或性能下降之前对其进行检测。

- Kubernetes 集群状态指标 — 监控 Kubernetes 对象的运行状况和状态，包括 Pod、节点、部署和服务。跟踪 pod 状态以识别 pod 处于待处理、失败或崩溃循环状态的 pod。监控节点状况以检测磁盘压力、内存压力或网络不可用等问题。使用 kubectl 或监控工具持续检查这些指标，并在 Pod 启动失败或节点不可调度时设置警报。
- 训练管道执行指标-跟踪成功和失败的管道运行、作业持续时间、步骤完成时间和编排错误。监控训练作业是否在预期的时间窗口内完成，以及失败率是否会随着时间的推移而增加。跟踪工作成功率、平均作业持续时间和失败时间等指标。这些指标可帮助您确定基础架构问题、配置问题或数据质量问题是否会导致训练失败。
- AWS 服务指标 — 跟踪支持 EKS 基础设施和培训工作负载的 AWS 服务的指标。监控 S3 指标，例如请求延迟、错误率和吞吐量，以确保数据加载性能保持一致。跟踪 EBS 卷指标，包括 IOPS、吞吐量和队列长度，以检测存储瓶颈。监控 VPC 流日志和网络指标，以确定节点之间或与外部服务的连接问题。
- Kubernetes 控制平面指标 — 监控 API 服务器、调度器、控制器管理器和 etcd 数据库，以检测影响集群操作的性能问题或故障。跟踪 API 服务器请求延迟、请求速率和错误率，确保控制平面快速响应调度请求。监控 etcd 数据库大小、提交持续时间和领导者更改，以检测稳定性问题。高的 API 服务器延迟或频繁的 etcd 领导者更改可能会延迟 pod 调度并延长训练作业的启动时间。

### 应用程序和实例日志

收集和分析应用程序和实例日志，以诊断仅靠指标无法解释的问题。日志提供了有关错误、状态更改和系统事件的详细背景信息，可帮助您了解训练作业失败或表现不佳的原因。将日志与指标关联可让您更快地查明根本原因。

- 应用程序日志-从训练作业、数据管道和机器学习框架中收集应用程序日志，以识别瓶颈并诊断故障。这些日志捕获有关任务执行的详细信息，包括数据加载错误、模型初始化失败、检查点保存错误和特定于框架的警告。将日志时间戳与指标峰值关联起来，以了解导致性能下降或故障的原因。例如，如果 GPU 利用率突然下降，请检查应用程序日志中是否存在表明数据管道停止或预处理失败的错误。使用 CloudWatch Logs 或 Fluent Bit 等集中式日志工具汇总来自所有 pod 的日志并使其可搜索。
- 实例日志-收集实例级日志，例如系统日志和 dmesg 输出，以检测硬件问题和内核级问题。这些日志揭示了 GPU 驱动程序错误、内存分配故障、磁盘 I/O 错误和网络接口问题等问题，这些问题可能不会出现在应用程序日志中。将实例日志与应用程序日志和指标关联起来，以确定训练失败是由硬件

问题还是应用程序问题引起的。例如，如果训练作业因内存不足错误而失败，请查看 dmesg 日志中是否有内核 OOM 杀手级消息，这些消息表明系统是否耗尽了内存或者应用程序是否超出了其容器限制。针对严重的硬件错误（例如 GPU XID 错误或磁盘故障）设置警报，这样您就可以在故障实例造成大规模训练中断之前将其替换。

以下各节展示了如何使用两种 AWS-recommended 方法收集上述指标：CloudWatch 容器见解和亚马逊托管 Prometheus Amazon Managed Prometheus 搭载亚马逊托管 Grafana 的 P rometh eus。如果您更喜欢设置最少且预建仪表板的 AWS-native 工具，请选择 Contain CloudWatch er Insights。如果您需要自定义控制面板、高级可视化功能或想要与现有监控基础设施集成，请选择搭载 Amazon Managed Grafana 的 Amazon Managed Prometheus。 Prometheus-based 有关可用容器洞察指标的完整列表，请参阅 Amazon EKS 和 Kubernetes 容器洞察指标。

## 考虑监控 Real-time 在线推理指标

在实时系统中，低延迟对于向用户或其他依赖系统提供及时响应至关重要。高延迟会降低用户体验或违反性能要求。影响推理延迟的组件包括模型加载时间、预处理时间、实际预测时间、后处理时间、网络传输时间。我们建议监控推理延迟，以确保满足服务级别协议 (SLA) 的低延迟响应，并针对以下内容制定自定义指标。在预期负载下进行测试，包括网络延迟、考虑并发请求以及使用不同的批量大小进行测试。

- 第一个令牌的时间 (TTFT)-从用户提交请求到收到回复的开头（第一个单词、令牌或块）的时间长度。例如，在聊天机器人中，你需要检查在用户提问后生成第一条输出（令牌）需要多长时间。
- End-to-End 延迟-这是从收到请求到返回响应的总时间。例如，测量从请求到响应的时间。
- 每秒输出令牌 (TPS)-表示模型开始响应后生成新标记的速度。例如，在聊天机器人中，你可以跟踪基线文本的语言模型的生成速度。
- 错误率-跟踪失败的请求，这可能表明存在性能问题。例如，监控对大型文档或某些字符的失败请求。
- 吞吐量-测量系统在单位时间内可以处理的请求或操作的数量。例如，跟踪每秒请求以处理峰值负载。

K/V (Key/Value) 缓存可以成为推理延迟的强大优化技术，尤其与基于变换器的模型相关。 K/V 缓存存储先前转换层计算中的键和值张量，从而减少自回归推理期间的冗余计算，尤其是在大型语言模型 (LLM) 中。缓存效率指标（专门用于 K/V 会话缓存的使用）：

- 缓存 hit/miss 比率 — 对于利用缓存（K/V 或嵌入缓存）的推理设置，请测量缓存起作用的频率。低命中率可能表示缓存配置或工作负载变化不佳，这两者都会增加延迟。

在随后的主题中，我们将演示如何收集上述几个指标的数据。我们将举例说明两种 AWS 推荐的方法：AWS-native CloudWatch 容器见解和使用亚马逊托管 Grafana 的开源亚马逊托管 Prometheus。您可以根据自己的整体可观测性需求选择其中一种解决方案。有关容器洞察指标的完整列表，请参阅 Amazon EKS 和 Kubernetes 容器洞察指标。

## 跟踪 GPU 内存使用情况

如the section called “考虑监控核心训练& Fine-Tuning 指标”主题中所述，GPU 内存使用对于防止内存不足 (OOM) 错误和确保高效利用资源至关重要。以下示例演示如何对训练应用程序进行检测以显示自定义直方图指标gpu_memory_usage_bytes，以及如何计算 P95 内存使用量以确定峰值消耗。请务必在暂存环境中使用样本训练作业（例如，微调变压器模型）进行测试。

### AWS-Native CloudWatch 容器见解示例

此示例演示了如何使用该 AWS-native 方法对训练应用程序进行校gpu_memory_usage_bytes验，使其显示为直方图。请注意，您的 AI/ML 容器必须配置为以 CloudWatch 嵌入式指标格式 (EMF) 格式发出结构化日志。 CloudWatch 日志解析 EMF 并发布指标。在训练应用程序中使用 aws_embedded_metrics 将 EMF 格式的结构化日志发送到日志， CloudWatch 日志会提取 GPU 指标。

```
from aws_embedded_metrics import metric_scope
import torch
import numpy as np

memory_usage = []

@metric_scope
def log_gpu_memory(metrics):
    # Record current GPU memory usage
    mem = torch.cuda.memory_allocated()
    memory_usage.append(mem)

    # Log as histogram metric
    metrics.set_namespace("MLTraining/GPUMemory")
    metrics.put_metric("gpu_memory_usage_bytes", mem, "Bytes", "Histogram")

    # Calculate and log P95 if we have enough data points
    if len(memory_usage) >= 10:
        p95 = np.percentile(memory_usage, 95)
        metrics.put_metric("gpu_memory_p95_bytes", p95, "Bytes")
        print(f"Current memory: {mem} bytes, P95: {p95} bytes")
```

```
# Example usage in training loop
for epoch in range(20):
    # Your model training code would go here
    log_gpu_memory()
```

### Prometheus 和 Grafana 示例

此示例演示如何使用 Python 中的 Prometheus 客户端库对训练应用程序进行检测，使其显示gpu_memory_usage_bytes`为直方图。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import pynvml

start_http_server(8080)
memory_usage = Histogram(
    'gpu_memory_usage_bytes',
    'GPU memory usage during training',
    ['gpu_index'],
    buckets=[1e9, 2e9, 4e9, 8e9, 16e9, 32e9]
)

# Function to get GPU memory usage
def get_gpu_memory_usage():
    if torch.cuda.is_available():
        # Get the current GPU device
        device = torch.cuda.current_device()

        # Get memory usage in bytes
        memory_allocated = torch.cuda.memory_allocated(device)
        memory_reserved = torch.cuda.memory_reserved(device)

        # Total memory usage (allocated + reserved)
        total_memory = memory_allocated + memory_reserved

        return device, total_memory
    else:
        return None, 0

# Get GPU memory usage
gpu_index, memory_used = get_gpu_memory_usage()
```

## 跟踪 Real-Time 在线推理的推理请求持续时间

如the section called “考虑监控核心训练& Fine-Tuning 指标”主题中所述，低延迟对于向用户或其他依赖系统提供及时响应至关重要。以下示例说明如何跟踪推理应用程序公开的自定义直方图指标。`inference_request_duration_seconds`计算第 95 个百分位 (P95) 延迟以专注于最坏的情况，在暂存环境中使用合成推理请求（例如，通过 Locust）进行测试，并设置警报阈值（例如 >500 毫秒）以检测 SLA 违规行为。

AWS-Native CloudWatch 容器见解示例

此示例演示了如何使用 AWS 嵌入式指标格式在推理应用程序中为 inference_request_duration_seconds 创建自定义直方图指标。 CloudWatch

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_inference_duration(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/Inference")
    metrics.put_metric("inference_request_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [0.1, 0.5, 1, 2, 5])

@metric_scope
def process_inference_request(metrics: MetricsLogger):
    start_time = time.time()

    # Your inference processing code here
    # For example:
    # result = model.predict(input_data)

    duration = time.time() - start_time
    log_inference_duration(metrics, duration)

    print(f"Inference request processed in {duration} seconds")

# Example usage
process_inference_request()
```

Prometheus 和 Grafana 示例

此示例演示了如何使用 Python 中的 Prometheus 客户端库在推理应用程序中为 ference_request_duration_seconds 创建自定义直方图指标：

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
request_duration = Histogram(
    'inference_request_duration_seconds',
    'Inference request latency',
    buckets=[0.1, 0.5, 1, 2, 5]
)
start_time = time.time()
# Process inference request
request_duration.observe(time.time() - start_time)
```

在 Grafana 中，使用`histogram_quantile(0.95, sum(rate(inference_request_duration_seconds_bucket[5m])) by (le, pod))`查询可视化 P95 延迟趋势。要了解更多信息，请参阅 Prometheus 直方图文档和 Prometheus 客户文档。

## 跟踪代币吞吐量以进行 Real-Time 在线推理

如the section called “考虑监控核心训练& Fine-Tuning 指标”主题中所述，我们建议监控代币处理时间，以衡量模型性能并优化扩展决策。以下示例说明如何跟踪推理应用程序公开的自定义直方图指标。`token_processing_duration_seconds`计算第 95 个百分位数 (P95) 持续时间以分析处理效率，在非生产集群中使用模拟的请求负载（例如 100 到 1000 requests/second）进行测试，并调整 KEDA 触发器以优化扩展。

AWS-Native CloudWatch 容器见解示例

此示例演示如何使用 AWS 嵌入式指标格式在推理应用程序中为 token_processing_duration_seconds 创建自定义直方图指标。 CloudWatch 它使用维度（`set_dimension ） with a custom `get_duration_bucket 函数将持续时间归类为存储桶（例如，“0.01”、“>1”）。

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')
```

```
@metric_scope
def log_token_processing(metrics: MetricsLogger, duration: float, token_count: int):
    metrics.set_namespace("ML/TokenProcessing")
    metrics.put_metric("token_processing_duration_seconds", duration, "Seconds")
    metrics.set_dimension("ProcessingBucket", get_duration_bucket(duration))
    metrics.set_property("TokenCount", token_count)

def get_duration_bucket(duration):
    buckets = [0.01, 0.05, 0.1, 0.5, 1]
    for bucket in buckets:
        if duration <= bucket:
            return f"<={bucket}"
    return f">{buckets[-1]}"

@metric_scope
def process_tokens(input_text: str, model, tokenizer, metrics: MetricsLogger):
    tokens = tokenizer.encode(input_text)
    token_count = len(tokens)

    start_time = time.time()
    # Process tokens (replace with your actual processing logic)
    output = model(tokens)
    duration = time.time() - start_time

    log_token_processing(metrics, duration, token_count)
    print(f"Processed {token_count} tokens in {duration} seconds")
    return output
```

### Prometheus 和 Grafana 示例

此示例演示了如何使用 Python 中的 Prometheus 客户端库在推理应用程序中为 token_processing_duration_seconds 创建自定义直方图指标。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
token_duration = Histogram(
    'token_processing_duration_seconds',
    'Token processing time per request',
    buckets=[0.01, 0.05, 0.1, 0.5, 1]
```

```
)
start_time = time.time()
# Process tokens
token_duration.observe(time.time() - start_time)
```

在 Grafana 中，使用`histogram_quantile(0.95, sum(rate(token_processing_duration_seconds_bucket[5m])) by (le, pod))`查询可视化 P95 处理时间趋势。要了解更多信息，请参阅 Prometheus 直方图文档和 Prometheus 客户文档。

测量检查点恢复延迟

如the section called “考虑监控核心训练& Fine-Tuning 指标”主题所述，检查点延迟是模型生命周期多个阶段的关键指标。以下示例说明如何跟踪您的应用程序公开的自定义直方图指标。`checkpoint_restore_duration_seconds`计算第 95 个百分位数 (P95) 持续时间以监控恢复性能，在非生产集群中使用 Spot 中断进行测试，并设置警报阈值（例如 <30 秒）以检测延迟。

AWS-Native CloudWatch 容器见解示例

此示例演示了如何使用 Insights 对批处理应用程序进行检测，以将 checkpoint_restore_duration_seconds 显示为直方图： CloudWatch

```
import boto3
import time
import torch
from aws_embedded_metrics import metric_scope, MetricsLogger

@metric_scope
def log_checkpoint_restore(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/ModelOperations")
    metrics.put_metric("checkpoint_restore_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [1, 5, 10, 30, 60])
    metrics.set_property("CheckpointSource", "s3://my-bucket/checkpoint.pt")

@metric_scope
def load_checkpoint(model, checkpoint_path: str, metrics: MetricsLogger):
    start_time = time.time()

    # Load model checkpoint
    model.load_state_dict(torch.load(checkpoint_path))
```

```
    duration = time.time() - start_time
    log_checkpoint_restore(metrics, duration)

    print(f"Checkpoint restored in {duration} seconds")
```

Prometheus 和 Grafana 示例

此示例演示如何使用 Python 中的 Prometheus `checkpoint_restore_duration_seconds` 客户端库对批处理应用程序进行检测，使其显示为直方图：

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import torch

start_http_server(8080)
restore_duration = Histogram(
    'checkpoint_restore_duration_seconds',
    'Time to restore checkpoint',
    buckets=[1, 5, 10, 30, 60]
)
with restore_duration.time():
    model.load_state_dict(torch.load("s3://my-bucket/checkpoint.pt"))
```

在 Grafana 中，使用`histogram_quantile(0.95, sum(rate(checkpoint_restore_duration_seconds_bucket[5m]) by (le))`查询可视化 P95 恢复延迟趋势。要了解更多信息，请参阅 Prometheus 直方图文档和 Prometheus 客户文档。

# 应用程序扩展和性能

> **Tip**
>
> 注册参加即将举行的 Amazon EKS AI/ML 研讨会。

## 管理 ML 构件、提供框架和启动优化

在 Amazon EKS 上部署机器学习 (ML) 模型需要仔细考虑如何将模型集成到容器映像和运行时环境中。这确保了可扩展性、可重复性和高效的资源利用率。本主题介绍了处理机器学习模型工件、选择服务框架以及通过预缓存等技术优化容器启动时间的不同方法，所有这些方法都是为减少容器启动时间而量身定制的。

## 在部署中处理 ML 模型构件

一个关键的决定是如何自己处理机器学习模型工件（例如权重和配置）。选择会影响映像大小、部署速度、模型更新频率和操作开销。请注意，在提及存储 “模型” 时，我们指的是模型工件（例如训练过的参数和模型权重）。在 Amazon EKS 上处理机器学习模型工件有不同的方法。每种方法都有其权衡取舍，最佳选择取决于模型的大小、更新节奏和基础设施需求。请考虑以下方法，从最低到最受推荐：

- 将模型烘焙到容器镜像中：在镜像构建过程中，将模型文件（例如.safetensors、.pth、.h5）复制到容器镜像（例如 Dockerfile）中。该模型是不可变图像的一部分。对于更新频率较低的小型号，我们建议使用这种方法。这可以确保一致性和可重复性，不提供加载延迟，简化依赖关系管理，但会导致图像大小变大，构建和推送速度变慢，需要重新构建和重新部署才能更新模型，而且由于注册表拉取吞吐量，因此不适合大型模型。
- 在运行时下载模型：在容器启动时，应用程序通过初始化容器或入口点中的脚本从外部存储（例如 Amazon S3，由 S3 CRT 支持，使用适用于 S3 CSI 驱动程序的 Mountpoint、AWS S3 CLI 或 s5cmd OSS CLI 等方法）下载模型，以优化高吞吐量传输。对于经常更新的大型模型，我们建议从这种方法开始。这样可以专注于容器镜像 code/runtime，无需重建即可轻松更新模型，支持通过存储元数据进行版本控制，但它会引入潜在的网络故障（需要重试逻辑），需要身份验证和缓存。

要了解更多信息，请参阅 EKS 人工智能研讨会中的加速拉取流程。

## 为机器学习模型提供服务

在 Amazon EKS 上部署和提供机器学习 (ML) 模型需要选择适当的模型服务方法来优化延迟、吞吐量、可扩展性和操作简便性。选择取决于您的模型类型（例如语言、视觉模型）、工作负载需求（例如实时推理）和团队专业知识。常见的方法包括原型设计 Python-based 设置、用于生产级功能的专用模型服务器以及用于实现高性能和效率的专用推理引擎。每种方法都需要权衡设置复杂性、性能和资源利用率。请注意，由于依赖关系，服务框架可能会增加容器映像的大小（多 GB），这可能会影响启动时间，请考虑使用构件处理技术进行解耦来缓解这种情况。从最少到最推荐的选项按推荐顺序列出：

使用 Python 框架（例如 FastAPI、 HuggingFace Transformers with PyTorch）使用 Python 框架开发自定义应用程序，将模型文件（权重、配置、标记器）嵌入到容器化节点设置中。

- 优点：原型设计简单，无需额外的基础架构， Python-only 与所有 HuggingFace 模型兼容，Kubernetes 部署简单。
- 缺点：仅限于单 request/simple 批处理、令牌生成缓慢（没有经过优化的内核）、内存效率低下 scaling/monitoring、缺少内存且启动时间长。
- 建议：用于需要自定义逻辑集成的初始原型设计或单节点任务。

使用专用的模型服务框架（例如 TensorRT-LLM，TGI）采用专门的服务器（如 TensorRT-LLM 或 TGI）进行机器学习推理，管理模型加载、路由和优化。这些支持诸如 safetensors 之类的格式，以及可选的编译或插件。

- 优点：提供批处理（static/in飞行或连续）、量化（INT8、FP8、GPTQ）、硬件优化（NVIDIA、AMD、英特尔、推理）和多 GPU 支持（并行度）。Tensor/Pipeline TensorRT-LLM 支持多种模型（LLM、 Encoder-Decoder），而 TGI 则利用 HuggingFace 集成。
- 缺点： TensorRT-LLM 需要编译而且是 NVIDIA-only；TGI 的批处理效率可能较低；两者都会增加配置开销，并且可能不适合所有模型类型（例如，非变压器）。
- 建议：适用于需要生产能力（例如 A/B 测试或具有兼容硬件的高吞吐量）的 PyTorch/TensorFlow 型号。

使用专门的高吞吐量推理引擎（例如 vLLM）利用高级推理引擎，例如 vLLM，优化 LLM 服务、飞行中批处理和量化（INT8 PagedAttention、、AWQ），可与 EKS 自动缩放集成。 FP8-KV

- 优点：高吞吐量和内存效率（可节省 40-60% 的 VRAM）、动态请求处理、令牌流、单节点 Tensor 并行多 GPU 支持以及广泛的硬件兼容性。
- 缺点：针对仅限解码器的变压器（例如 LLaMA）进行了优化，对非变压器型号效果较差，需要兼容的硬件（例如 NVIDIA GPU）和设置工作。
- 建议：在 EKS 上进行大容量、低延迟 LLM 推理的首选，可最大限度地提高可扩展性和性能。

## 优化容器镜像提取时间

大型容器镜像可能会导致冷启动延迟，从而影响 pod 的启动延迟。对于延迟敏感型工作负载，例如水平扩展的实时推理工作负载，Pod 的快速启动至关重要。考虑以下方法来优化容器镜像的提取时间：

### 缩小容器镜像大小

在启动期间减小容器镜像的大小是缩小镜像的另一种方法。您可以在容器镜像构建过程的每个步骤中进行削减。首先，请选择包含所需依赖项最少的基础映像。在映像构建过程中，仅包含必需的基本库和工件。构建图像时，请尝试组合多个RUN或COPY命令来创建较少数量的较大图层。对于 AI/ML 框架，使用多阶段构建来分隔构建和运行时，仅复制所需的工件（例如，注册表或本地上下文的 viaCOPY –from=），并选择诸如仅限运行时的图像之类的变体（例如，`pytorch/pytorch:2.7.1-cuda11.8-cudnn9-runtime`3.03 GB 与 devel 为 6.66 GB）。要了解更多信息，请参阅 EKS 人工智能研讨会中的缩小容器映像大小。

## 减少图像拉取延迟

对于在私有子网中运行的 EKS 集群，请为 Amazon ECR 配置 VPC 接口终端节点，以保持 AWS 私有网络上的图像拉取流量，从而减少延迟并消除 NAT 网关数据处理费用。有关设置说明，请参阅为 Amazon ECR 创建 VPC 终端节点。

此外，如果您的工作负载依赖于来自外部注册表的上游容器映像，请考虑使用 ECR pull th rough cache 在私有 ECR 注册表中代理和缓存这些映像。

## 优化图像拉取的网络带宽

拉取大型容器镜像时，EKS 工作节点可用的网络带宽会直接影响训练和推理工作负载的启动时间。所有当前一代的 Nitro-based 实例都使用弹性网络适配器 (ENA)，但可用带宽因实例大小而有很大差异，因此了解基准带宽和突发带宽之间的区别至关重要（参见 Amazon EC2 实例网络带宽）。为了最大限度地缩短工作负载的准备时间，请选择具有足够基准网络带宽的实例大小（相对于您的映像大小和拉取并发性）。

## 使用 SOCI 快照器拍摄图像 Pre-pull

对于无法轻易最小化的超大图像，您可以使用在并行拉取和解压缩模式下配置的开源 Seekable OCI (SOCI) 快照器。此解决方案允许您使用现有映像，而无需重新构建或修改构建管道。当将具有非常大的映像的工作负载部署到高性能 EC2 计算实例时，此选项特别有效。它适用于高吞吐量网络和高性能存储配置，就像扩展 AI/ML 工作负载一样。

SOCI 并行 pull/unpack 模式通过可配置的并行化策略提高端到端图像拉取性能。更快的映像提取和准备速度直接影响您部署新工作负载和高效扩展集群的速度。图像拉取有两个主要阶段：

1。将图层从注册表提取到节点

为了优化图层抓取，SOCI 在每层创建多个并发 HTTP 连接，使下载吞吐量成倍超过单一连接限制。它将大层拆分成块，然后通过多个连接同时下载它们。这种方法有助于使您的可用网络带宽饱和，并显著缩短下载时间。这对于单层可能有几千兆字节 AI/ML 的工作负载特别有价值。

2。拆包并准备这些图层以创建容器

为了优化图层解包，SOCI 会同时处理多个图层。它使用可用的 CPU 内核同时解压缩和提取多个图层，而不是等待每层完全解压后再开始下一层。这种并行处理将传统的 I/O-bound 拆包阶段转变为可随可用内核进行扩展的 CPU-optimized 操作。系统会仔细协调这种并行化，以保持文件系统的一致性，同时最大限度地提高吞吐量。

SOCI parallel pull 模式使用双阈值控制系统，该系统具有用于下载并发性和解包并行性的可配置参数。这种精细控制允许您微调 SOCI 的行为，以满足您的特定性能要求和环境条件。了解这些参数有助于优化运行时间，以获得最佳的拉取性能。

参考

- 有关解决方案和调整权衡的更多信息，请参阅 SOCI 项目存储库中的功能文档。 GitHub
- 有关在 Amazon EKS 上使用 Karpenter 的动手示例，请参阅使用 SOCI snapshotter 并行模式的 Karpenter 蓝图。 pull/unpack
- 有关配置 Bottlerocket 进行并行拉动的信息，请参阅 Bottlerocket 中的 s oci-snapshotter Pallel Pull Unpack 模式 Documentation.o

## 将 EBS 快照用于映像 Pre-pull

您可以为缓存的容器映像拍摄 Amazon Elastic Block Store (EBS) 快照，然后在 EKS 工作节点上重复使用此快照。这样可以确保在节点启动时在本地预取图像，从而缩短 pod 初始化时间。有关使用 Karpenter 和 EKS Terraform Blueprints 管理托管节点组的更多信息，请参阅使用 Bottlerocket 数据量缩短亚马逊 E KS 上的容器启动时间。

要了解更多信息，请参阅 EKS 人工智能研讨会中的使用 containerd 快照和使用 EBS 快照将容器映像预加载到 Bottlerocket 数据卷中。

## 使用容器运行时缓存存储 Pre-pull 镜像

你可以使用 Kubernetes 资源（例如 DaemonSet 或 Deployment）将容器镜像预拉到节点上，以填充节点的容器运行时缓存。容器运行时缓存是由容器运行时管理的本地存储（例如，c ontainerd），从注册表中提取图像后存储在那里。 Pre-pulling 确保图像在本地可用，从而避免 pod 启动期间的下载延迟。 当图像经常更改（例如频繁更新）、未预先配置 EBS 快照时、构建 EBS 卷比直接从容器注册表中提取更耗时，或者节点已经在集群中并且需要使用多个可能的映像之一按需启动 pod 时，这种方法特别有用。

Pre-pulling 无论需要哪种映像，所有变体都可确保快速启动。例如，在需要使用 10 种不同技术构建 100,000 个小型模型的大规模并行机器学习工作负载中，通过在大型集群（例如数千 DaemonSet 个节点）中预先拉取 10 张图像可最大限度地缩短 pod 启动时间，从而避免按需拉取，从而在不到 10 秒的时间内完成。使用容器运行时缓存方法无需管理 EBS 快照，可确保您始终使用最新的容器映像版本，但是对于节点可扩展 in/out的实时推理工作负载 DaemonSets，由 Cluster Autoscaler 等工具添加的新节点可能会在预 DaemonSet 拉取完成映像拉取之前调度工作负载 pod。无论如何，这都可能导致新节点上的初始 Pod 触发拉动，从而可能延迟启动并影响低延迟要求。此外，当磁盘使用率超过特定阈

值或超过配置的最大未使用年限时，kubelet 图像垃圾回收会通过删除未使用的图像来影响预拉取的图像。在缩放in/out 模式中，这可能会驱逐空闲节点上的图像，这需要在随后的扩展过程中重新拉取，从而降低缓存对于突发性工作负载的可靠性。

有关将图像预拉入容器运行时缓存的示例，请参阅 AWS GitHub 存储库。

## 考虑将 NVMe 用于 kubelet 和 containerd 存储

考虑配置kubelet并使用containerd临时 NVMe 实例存储磁盘以获得更高的磁盘性能。容器拉取过程包括从注册表下载容器镜像，然后将其层解压缩为可用格式。要优化解压缩期间的 I/O 操作，您应该评估哪些方法可以为容器主机的实例类型提供更高的 I/O 性能和吞吐量：具有本地存储的 NVMe 支持实例与 EBS 卷的实例。 IOPS/throughput对于具有 NVMe 本地存储的 EC2 实例，可以考虑为 kubelet () /var/lib/kubelet、containerd () 和 Pod logs (/var/lib/containerd) 配置节点的底层文件系统，以使用临时 NVMe 实例存储磁盘来实现更高的性能和吞吐量。/var/log/pods I/O

节点的临时存储空间可以在请求临时存储的 Pod 和下载到节点的容器镜像之间共享。如果将 Karpenter 与 Bottlerocket 或 AL2023 EKS 优化的 AMI 一起使用，则可以在中EC2NodeClass进行配置，方法是将StorePolicy 实例设置为 RAID0，或者如果使用托管节点组，则可以将StoragePolicy 本地NodeConfig设置为用户数据的一部分。 AL2023

# 为本指南做出贡献

任何人都可以为最佳实践指南做出贡献。《EKS 最佳实践指南》的 AsciiDoc 格式为 GitHub。

## 现有贡献者摘要

- bpg-docs.code-workspace使用 VS Code 打开，即可自动安装 AsciiDoc 扩展程序。
  - 在 Visual Studio Marketplace 上了解有关AsciiDoc 扩展的更多信息。
- AWS Docs 网站的源文件存储在 latest/bpg
- 其语法与 markdown 非常相似。
  - 查看 AsciiDoctor 文档中的语法参考。
- 仅部署latest/bpg/images文档平台。每个指南章节都有一个指向该目录的符号链接。例如，latest/bpg/networking/images指向latest/bpg/images。

## 设置本地编辑环境

如果您计划经常编辑指南，请设置本地编辑环境。

### 分叉并克隆存储库

您需要熟悉gitgithub、和文本编辑器。有关git和入门的信息github，请参阅 GitHub 文档中的 GitHub 账户入门。

1. 在上查看 EKS 最佳实践指南 GitHub。
2. 创建项目存储库的分支。在 GitHub 文档中学习如何分叉存储库。
3. 克隆您的项目仓库分支。了解如何克隆您的分叉存储库。

### 打开 VS 代码工作区

AWS 建议使用微软提供的 Visual Studio 代码来编辑该指南。有关 VS Code 的更多信息，请参阅 V isual Studio 代码文档中的下载 V isual Studio 代码和开始使用 Visual Studio 代码。

1. 打开 VS Code。

2. 从克隆的存储库中打开bpg-docs.code-workspace文件。
3. 如果这是您第一次打开此工作区，请接受安装 AsciiDoc 扩展程序的提示。此扩展程序会检查 AsciiDoc 文件的语法并生成实时预览。
4. 浏览到该latest/bpg目录。此目录包含部署到 AWS 文档站点的源文件。源文件按指南部分进行组织，例如 “安全” 或 “网络”。

## 编辑文件

1. 在编辑器中打开一个文件。
   - 查看 AsciiDoc 语法以了解如何创建标题、链接和列表。
   - 您可以使用 Markdown 语法来设置文本格式、创建列表和标题。你不能使用 Markdown 语法来创建链接。
2. 打开页面的实时预览。
   - 首先，按ctrl-k或cmd-k（视键盘而定）。其次，按v。这将在分屏视图中打开预览。

AWS 建议使用功能分支来组织您的更改。学习如何使用 git 创建分支。

## 提交拉取请求

您可以从 GitHub 网站或 GitHub cli 创建拉取请求。

了解如何使用 GitHub 网站从 fork 创建拉取请求。

了解如何使用 GitHub cli 创建拉取请求。

# 使用 github.dev 基于网络的编辑器

github.dev基于 Web 的编辑器基于 VS Code。这是无需任何设置即可编辑多个文件和预览内容的好方法。

它支持该 AsciiDoc 扩展。你可以使用 GUI 进行 git 操作。基于 Web 的编辑器没有用于运行命令的 shell 或终端。

你必须有一个 GitHub 账户。如果需要，系统将提示您登录。

# 启动 GitHub 基于 Web 的编辑器。

# 编辑单个页面

您可以使用快速更新各个页面 GitHub。每个页面的底部都包含一个 “# 在 GitHub” 上编辑此页面” 链接。

1. 导航到本指南中您要编辑的页面
2. 点击底部的 “编辑此页面 GitHub” 链接
3. 单击 GitHub 文件查看器右上角的编辑铅笔图标，或按 e
4. 编辑文件
5. 使用 “提交更改...” 按钮提交您的更改。此按钮可创建 GitHub 拉取请求。指南维护者将审查此拉取请求。审阅者将批准拉取请求或请求更改。

# 查看和设置页面的 ID

本页说明如何查看和设置页面 ID。

页面 ID 是一个唯一的字符串，用于标识文档网站上的每个页面。当你进入特定页面时，你可以在浏览器的地址栏中查看页面 ID。页面 ID 用于 URL、文件名和创建交叉引用链接。

例如，如果您正在查看此页面，则浏览器地址栏中的网址将类似于：

```
https://docs.aws.amazon.com/view-set-page-id.html
```

网址 (`view-set-page-id`) 的最后一部分是页面 ID。

## 设置页面 ID

创建新页面时，需要在源文件中设置页面 ID。页面 ID 应该是一个简洁的连字符串，用于描述页面内容。

1. 在文本编辑器中打开新页面的源文件。
2. 在文件顶部，添加以下行。它应该在第一个标题之上。

   ```
   [#my-new-page]
   ```

   `my-new-page`替换为新页面的页面 ID。

3. 保存该文件。

> **Note**
>
> 在整个文档网站中，页面 IDs 必须是唯一的。如果您尝试使用现有的页面 ID，则会收到构建错误。

# 创建新页面

了解如何创建新页面和更新指南目录。

## 创建页面元数据

1. 确定页面标题和页面简称。页面短标题是可选的，但如果页面标题超过几个字，则建议使用该标题。
2. 确定页面的 ID。这在《EKS 最佳实践指南》中必须是唯一的。惯例是全部使用小写字母，并使用-分隔单词。
3. 如果需要，在文件夹中创建一个新的 asciidoc 文件，然后向该文件中添加以下文本：

   Example

   [。” 主题 "] [#<page-id>] =<page-title>: info_titleabbrev: < > page-short-title

   例如，

   Example

   [。” 主题 "] [#scalability] = EKS 可扩展性最佳实践：info_titleabbrev：可扩展性

## 添加到目录

1. 在目录中打开父页面的文件。对于新的顶级指南章节，父文件为`book.adoc`。
2. 在父文件的底部，更新并插入以下指令：

   Example

   包括:: <new-filename>[leveloffset=+1]

例如，

Example

包括:: dataplane.adoc [leveloffset=+1]

# 插入图片

1. 查找您正在编辑的页面的图像前缀。查看文件标题中的`:imagesdir:`属性。举个例子，`:imagesdir: images/reliability/`
2. 将您的图片放在此路径中，例如 `latest/bpg/images/reliability`
3. 为您的图片确定合适的替代文本。为图像写一个简短的高级描述。例如，“具有三个可用区的 VPC 示意图” 是恰当的替代文本。
4. 使用替代文本和图像文件名更新以下示例。在所需位置插入。

Example

图片:: <image-filename>[< image-alt-text >]

例如，

Example

图片:: eks-data-plane-connectivity .jpeg [网络图]

# 向淡水河谷查询风格

1. 安装 Vale CLI。
2. 运行 `vale sync`
3. 从 Visual Studio Marketplace 安装谷扩展程序。
4. 重启 VS Code，然后打开一个 AsciiDoc 文件
5. VS Code 在有问题的文本下划线。在 VS Code 文档中学习如何处理错误和警告。

# 生成本地预览版

1. 在 Linux 或 macOS `brew` 上使用安装该`asciidoctor`工具
   - 在文档中学习如何安装 asciidoctor cli。 AsciiDoctor
   - 了解如何安装 brew 软件包管理器。
2. 打开终端，然后导航至 `latest/bpg/`
3. 运行 `asciidoctor book.adoc`
   - 查看所有语法警告和错误
4. 打开`book.html`输出文件。
   - 在 macOS 上，你可以运行`open book.html`在默认浏览器中打开预览。

# AsciiDoc 备忘单

## 基本格式化

```
*bold text*
_italic text_
`monospace text`
```

## 标头

```
= Document Title (Header 1)
== Header 2
=== Header 3
==== Header 4
===== Header 5
====== Header 6
```

## Lists

未排序的列表：

```
- Item 1
- Item 2
-- Subitem 2.1
-- Subitem 2.2
```

```
- Item 3
```

排序后的列表：

```
. First item
. Second item
.. Subitem 2.1
.. Subitem 2.2
. Third item
```

## 链接

```
External link:  https://example.com[Link text]
Internal link: <<page-id>>
Internal link: xref:page-id[Link text]
```

## 图片

```
image::image-file.jpg[Alt text]
```

## 代码块

```
 [source,python]
 ----
 def hello_world():
     print("Hello, World!")
 ----
```

## 表

了解如何构建基础表。

```
[cols="1,1"]
|===
|Cell in column 1, row 1
|Cell in column 2, row 1

|Cell in column 1, row 2
|Cell in column 2, row 2
```

```
|Cell in column 1, row 3
|Cell in column 2, row 3
|===
```

## 警戒插件

```
NOTE: This is a note admonition.

WARNING: This is a warning admonition.

TIP: This is a tip admonition.

IMPORTANT: This is an important admonition.

CAUTION: This is a caution admonition.
```

预览：

> **Note**
>
> 这是一个笔记警戒插件。

## 包含

```
include::filename.adoc[]
```

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。