检测和缓解灰色故障

高级多可用区弹性模式

高级多可用区弹性模式: 检测和缓解灰色故障

Table of Contents

高级多可用区弹性模式

发布日期：2023 年 7 月 11 日 (文档修订)

许多客户在可用性高的多可用区 (AZ) 配置中运行其工作负载。这些架构在二进制故障事件期间表现良好，但经常遇到灰色故障。这种故障的表现形式很微妙，无法快速而明确地对其进行检测。本文提供了有关如何检测工作负载的指导，以检测隔离到单个可用区的灰色故障的影响，然后采取措施减轻对该可用区的这种影响。

简介

本文档的目的是帮助您更有效地实现弹性多可用区架构。在 Amazon 虚拟私有云 (VPC) 网络中构建弹性系统的最佳实操之一是将每个工作负载都部署到多个可用区。

可用区是一个或多个具有冗余电源、网络和连接的离散数据中心。通过使用多个可用区，您可以获得比在单个数据中心中可用性、容错能力和可扩展性更高的工作负载。

许多 AWS 服务（例如 Amazon Elastic Compute Cloud (EC2) 自动扩缩或 Amazon Relational Database Service (Amazon RDS)都提供多可用区配置。这些服务不需要您额外构建任何可观测性或失效转移工具。它们使工作负载能够适应影响单个可用区的 AWS 区域 中易于检测到的二进制故障模式。它可能是完全的物理硬件故障、断电或影响大多数资源的潜在软件错误。

但是还有另一类故障，称为灰色故障，表现形式微妙，无法快速明确地对其进行检测。这反过来又会导致投入更长的时间来缓解故障所造成的影响。本文重点介绍灰色故障可能对多可用区架构产生的影响、如何检测它们，以及如何缓解这些影响。

> 本白皮书中提供的指南主要适用于具有以下特点的特定类别工作负载：
>
> - 主要使用区域 AWS 服务
> - 需要提高单个区域的弹性
> - 愿意进行大量投资来建立所需的可观测性和弹性模式
>
> 在这些工作负载中，您可能不愿意做出 ??? 中提出的部分或全部权衡，或者不具备使用多个区域的选项。这些类型的工作负载可能只占整个产品组合的一小部分，因此应在工作负载级别而不是平台级别考虑本指南。

灰色故障

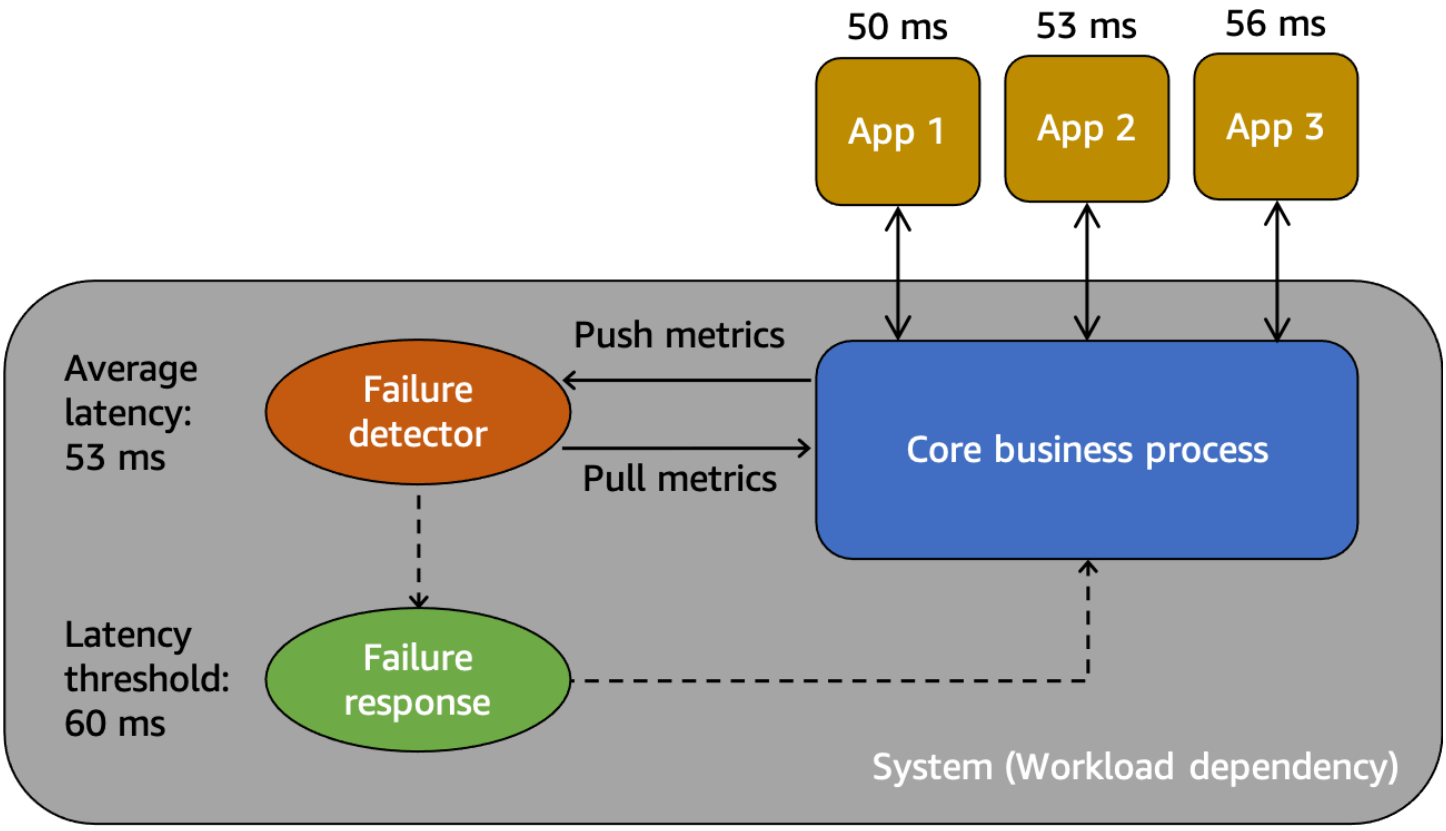
灰色故障因其差异可观测性的特征而得名，反映不同实体对故障的观察方式不同。下面我们来做一下定义。

差异可观测性

您操作的工作负载通常都有依赖项。比如，它们可以是您用来构建工作负载的 AWS 云服务，也可以是用于联合身份验证的第三方身份提供者 (IdP)。这些依赖项几乎会一直实现自己的可观测性：记录有关错误、可用性和延迟的指标，以及由其客户使用情况生成的其他指标。当其中一个指标超过阈值时，依赖项通常会采取一些措施来进行纠正。

这些依赖项通常有多个使用者使用其服务。使用者还可以实现自己的可观测性，并记录有关其与依赖项交互的指标和日志，记录诸如磁盘读取延迟、API 请求失败次数或数据库查询花了多长时间之类的信息。

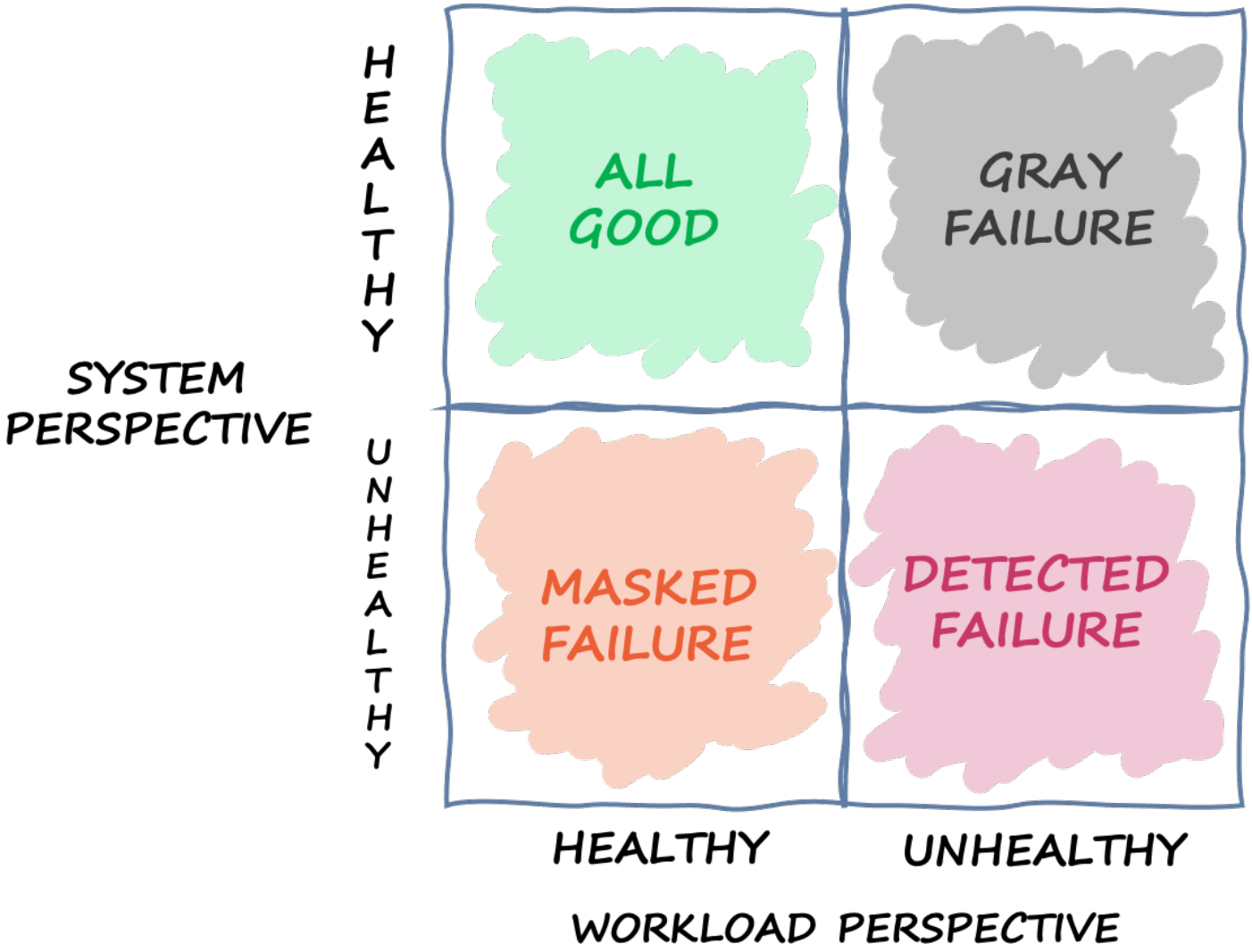
下图使用抽象模型描述了这些交互和测量。



用于了解灰色故障的抽象模型

首先，在此场景中，系统是使用者应用程序 1、应用程序 2 和应用程序 3 的依赖项。该系统具有故障探测器，可以检查从核心业务流程中创建的指标。它还具有故障响应机制，可以缓解或纠正故障探测器观察到的问题。我们可以看到，系统总体的平均延迟为 53 毫秒，并设置了阈值，在平均延迟超过 60 毫秒时调用故障响应机制。应用程序 1、应用程序 2 和应用程序 3 也对它们与系统的交互进行了自己的观察，记录的平均延迟分别为 50 毫秒、53 毫秒和 56 毫秒。

差异可观测性是指其中一个系统使用者检测到系统运行状况不佳，但系统自己的监控无法检测到问题或受到的影响未超过警报阈值的情况。假设应用程序 1 的平均延迟开始达到 70 毫秒，而不是 50 毫秒。而应用程序 2 和应用程序 3 的平均延迟时间没有变化。这会将底层系统的平均延迟增加到 59.66 毫秒，但这并没有超过激活故障响应机制的延迟阈值。但是，应用程序 1 的延迟增加了 40%。这可能会超过为应用程序 1 配置的客户端超时时间，从而影响其可用性，也可能在更长的交互链中造成级联影响。从应用程序 1 的角度来看，它所依赖的底层系统运行状况不佳，但是从系统本身以及应用程序 2 和应用程序 3 的角度来看，该系统是正常的。下图总结了这些不同的视角。



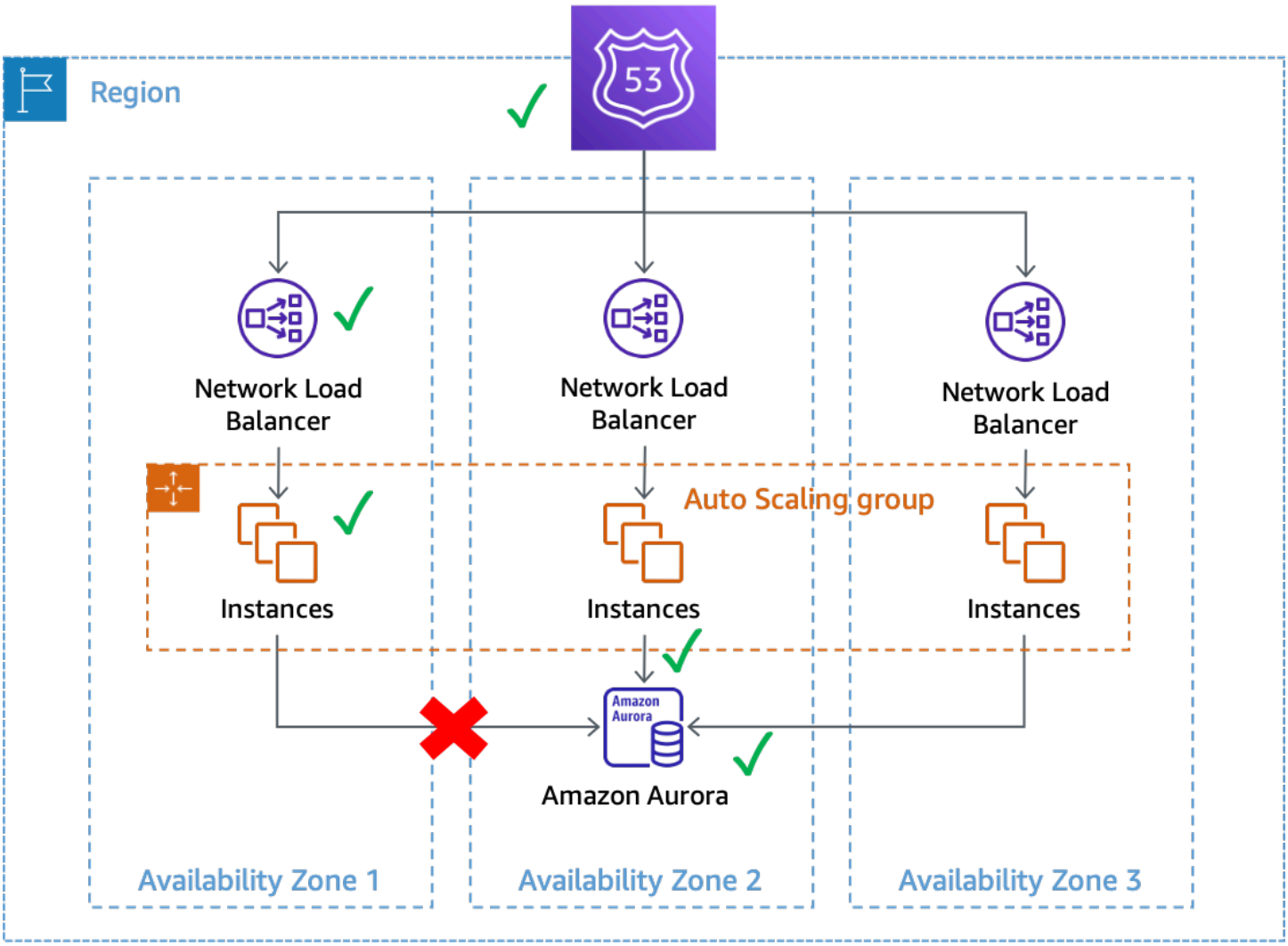
象限图定义了从不同视角来看系统可能处于的不同状态

故障也可能从一个象限跨越到另一个象限。事件可能最开始时是灰色故障，然后变成检测到的故障，然后转变为屏蔽故障，然后可能回到灰色故障。没有明确的周期，在根本原因得到解决之前，几乎随时有可能再次出现故障。

我们从中得出的结论是，工作负载不能总是依靠底层系统来检测和缓解故障。无论底层系统多么先进、有弹性，总有故障会逃脱检测，或保持在反应阈值以内。该系统的使用者（例如应用程序 1）需要得到强化，以快速检测和缓解灰色故障造成的影响。这就要求针对这些情况建立可观测性和恢复机制。

灰色故障示例

灰色故障可能会对 AWS 中的多可用区系统产生影响。例如，以部署在三个可用区的自动扩缩组中的 Amazon EC2 实例的实例集为例。它们都连接到一个可用区内的 Amazon Aurora 数据库。然后，出现灰色故障，会影响可用区 1 和可用区 2 之间的网络连接。这种损坏的结果是，来自可用区 1 中实例的新的和现有数据库连接中有一定比例会失败。下图展示了这种情况。



灰色故障影响了可用区 1 中实例的数据库连接

在此示例中，Amazon EC2 认为可用区 1 中的实例运行状况良好，因为它们仍可通过系统和实例状态检查。Amazon EC2 Auto Scaling 也不会检测到对任何可用区的直接影响，而会继续在已配置的可用区中启动容量。Network Load Balancer (NLB) 会根据针对 NLB 端点执行的 Route 53 运行状况检查，认为其后面的实例运行状况良好。同样，Amazon Relational Database Service (Amazon RDS) 也会认为数据库集群运行状况良好，不会触发自动失效转移。我们有许多不同的服务，它们都认为其服务和资源运行状况良好，但是工作负载检测到影响其可用性的故障。这是一个灰色故障。

应对灰色故障

如果您的 AWS 环境中出现灰色故障，您通常有三个选项可以选择：

- 什么都不做，等待损坏结束。

- 如果损坏隔离在单个可用区，请撤离该可用区。
- 失效转移到另一个 AWS 区域 并利用 AWS 区域隔离的优势来缓解影响。

对于大多数工作负载，许多 AWS 客户都认为第一个选项还不错。他们可以接受可能延长的恢复时间目标 (RTO)，只要不必构建额外的可观测性或弹性解决方案。其他客户选择实现第三个选项，即多区域灾难恢复 (DR)，作为其针对各种故障模式的缓解计划。在这些场景中，多区域架构可以很好地发挥作用。但是，使用这种方法时需要进行一些权衡（有关多区域注意事项的完整讨论，请参阅AWS 多区域基础知识）。

首先，构建和运营多区域架构具有挑战性、复杂性且可能产生高昂成本。要使用多区域架构，您需要认真考虑选择哪种灾难恢复策略。仅仅为了处理区域损坏而实施多区域主动-主动灾难恢复解决方案在财务上可能不可行，而备份和恢复策略可能无法满足您的弹性要求。此外，多区域失效转移必须要在生产中持续开展，这样您才能确信它们能在需要时起作用。所有这些都需要专门投入大量时间和资源来构建、操作和测试。

其次，如今使用 AWS 服务跨 AWS 区域 复制数据都是异步执行的。异步复制可能导致数据丢失。这意味着在区域失效转移期间，可能会出现一定程度的数据丢失和不一致。您对数据丢失量的容忍度即为恢复点目标 (RPO)。如果客户对数据一致性要求严格，则当主区域再次可用时，他们必须建立协调系统来修复这些一致性问题。或者，他们必须构建自己的同步复制或双写系统，这会对响应延迟、成本和复杂性产生重大影响。它们还将次要区域作为每项事务的硬依赖项，这可能会降低整个系统的可用性。

最后，对于许多使用主动/备用方法的工作负载，执行失效转移到另一个区域所需的时间不为零。您的工作负载组合可能需要按特定顺序在主区域中关闭，需要耗尽连接或停止特定进程。然后，可能需要按特定顺序恢复这些服务。在投入使用之前，新资源可能还需要进行配置或需要一段时间才能通过所需的运行状况检查。在此失效转移过程中，各种资源完全不可用。这就是 RTO 所关心的问题。

在一个区域内，许多 AWS 服务都提供高度一致的数据持久性。Amazon RDS 多可用区部署使用同步复制。Amazon Simple Storage Service (Amazon S3) 提供强大的先写后读一致性。Amazon Elastic Block Storage (Amazon EBS) 提供多卷崩溃一致性快照。Amazon DynamoDB 可以强一致性读取。与多区域架构相比，这些功能可以帮助您在单个区域中实现更低的 RPO（在大多数情况下为零 RPO）。

撤离可用区的 RTO 低于多区域策略，因为您已为各个可用区配置了基础设施和资源。当可用区受损时，多可用区架构可以继续以静态方式运行，而不必小心翼翼地订购被关闭的服务并进行备份，也不必耗尽连接。在可用区撤离期间，由于工作转移到剩余的可用区，许多系统可能只会出现轻微的降级，而不会像区域失效转移那样出现完全不可用时期。如果系统设计为能够在可用区故障时保持静态稳定（在本例中，这意味着在其他可用区预先配置容量以接纳负载），则工作负载的客户可能根本感受不到影响。

除了您的工作负载外，单个可用区的损坏还可能影响一项或多项 AWS 区域服务。如果您观察到区域影响，则应将该事件视为区域服务损坏，尽管该影响的来源位于单个可用区。撤离可用区并不能缓解此类问题。发生这种情况时，请使用您现有的应对计划来应对区域服务损坏。

本文档的其余部分重点介绍第二种选择，即撤出可用区，以此来降低单可用区灰色故障的 RTO 和 RPO。这些模式可以帮助多可用区架构实现更高价值和效率，对于大多数类别的工作负载，还可以减少创建多区域架构来处理这些类型事件的需求。

多可用区可观测性

为了能够在单个可用区发生隔离事件期间撤出可用区，首先您必须能够检测到故障已实际上隔离到单个可用区。这要求对系统在每个可用区中的行为具备高可见性。许多 AWS 服务都提供即时使用的指标，可针对您的资源提供运营见解。例如，Amazon EC2 提供了许多指标，例如 CPU 利用率、磁盘读写以及进出网络流量。

但是，随着您使用这些服务构建工作负载，需要了解的指标就不仅仅是这些标准指标了。您希望了解您的工作负载为客户提供了何种体验。此外，您的指标需要与生成这些指标的可用区保持一致。只有获得这些见解，您才能够检测到各种不同可见度的灰色故障。为此，您需要进行分析；

要进行分析，就需要编写显式代码。该代码应该执行各种工作，例如记录任务花费了多长时间、对成功或失败的项目计数、收集有关请求的元数据等等。您还需要提前定义阈值，以定义哪些是正常情况以及哪些是异常情况。您应该针对工作负载中的延迟、可用性和错误计数制定目标并规定不同严重程度阈值。Amazon Builders' Library 的文章分析分布式系统来获得操作可视性提供了许多最佳实操。

指标既应从服务器端生成，也应从客户端生成。生成客户端指标和了解客户体验的最佳实操是使用金丝雀，该软件可以定期探测您的工作负载和记录指标。

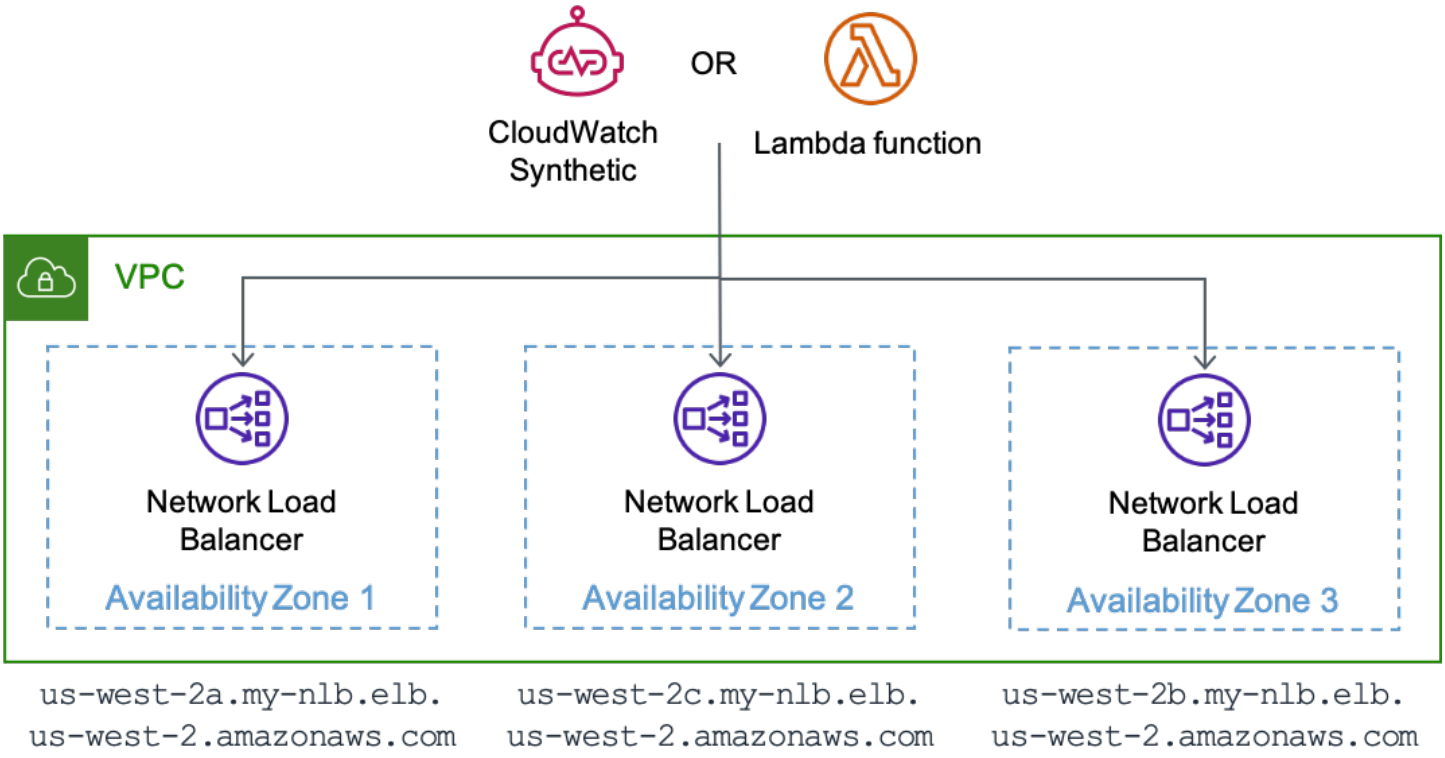
除了生成这些指标外，您还需要了解其上下文。执行此操作的一种方法是使用维度。维度为指标提供了唯一的身份，并有助于您了解这些指标传达的信息。对于那些用于识别工作负载中故障的指标（例如，延迟、可用性或错误数），您需要使用与故障隔离边界一致的维度。

例如，如果您使用模型视图控制器 (MVC) Web 框架在一个区域的多个可用区中运行 Web 服务，则应使用 `Region`、`Availability Zone ID`、`Controller`、`Action` 和 `InstanceId` 维度集（如果您在使用微服务，则可以使用服务名称和 HTTP 方法，而非控制器和操作名称）。这样做，是因为您希望使用这些边界隔离不同类型的故障，而不希望 Web 服务代码中本来只会影响产品展列的错误对主页产生影响。同样，您也不希望看到，单个 EC2 实例上的完整 EBS 卷会影响其他 EC2 实例为您的 Web 内容提供支持。可用区 ID 维度使您可以通过一致的方式确定可用区在不同 AWS 账户 中的影响。您可以通过多种不同的方式在工作负载中找到可用区 ID。请参阅附录 A – 获取可用区 ID ，了解一些示例。

本文档的示例中主要使用 Amazon EC2 作为计算资源，如果您使用 Amazon Elastic Container Service (Amazon ECS) 和 Amazon Elastic Kubernetes Service (Amazon EKS) 计算资源作为您的维度组件，则可以将示例中的 `InstanceId` 替换为相应的容器 ID。

如果您的工作负载有区域端点，则您的金丝雀还可以在其指标中使用 `Controller`、`Action`、`AZ-ID` 和 `Region` 维度。如果这样，请调整您的金丝雀，使其在其测试的可用区中运行。这样可以确保，如果金丝雀正在测试的某个可用区受到隔离的可用区事件的影响，金丝雀记录的指标不会因此使其测试的

其他可用区看起来有异常。例如，您的金丝雀可以利用区域 DNS 名称测试借助网络负载均衡器 (NLB) 或应用程序负载均衡器 (ALB) 运行的服务的每个区域端点。



在 CloudWatch Synthetics 或 AWS Lambda 函数上运行的金丝雀正在测试 NLB 的各个区域端点

通过生成具有这些维度的指标，您可以建立 Amazon CloudWatch 警报，这样当这些边界内的可用性或延迟发生变化时，这些警报会通知您。您也可以使用控制面板快速分析这些数据。为了高效使用指标和日志，Amazon CloudWatch 提供了嵌入式指标格式 (EMF)，使您能够在日志数据中嵌入自定义指标。CloudWatch 会自动提取自定义指标，以便您对其进行可视化处理并设置警报。AWS 为不同的编程语言提供了多个客户端库，让您可以轻松开始使用 EMF。它们可以与 Amazon EC2、Amazon ECS、Amazon EKS、AWS Lambda 和本地环境一起使用。通过将指标嵌入到日志中，您还可以使用 Amazon CloudWatch Contributor Insights 来创建显示贡献者数据的时间序列图表。这样，我们就可以显示按维度（如 `AZ-ID`、`InstanceId` 或 `Controller`）和日志中的任何其他字段（如 `SuccessLatency` 或 `HttpResponseCode`）分组的数据。

```
{
  "_aws": {
    "Timestamp": 1634319245221,
    "CloudWatchMetrics": [
      {
        "Namespace": "workloadname/frontend",
        "Metrics": [
```

```
          { "Name": "2xx", "Unit": "Count" },
          { "Name": "3xx", "Unit": "Count" },
          { "Name": "4xx", "Unit": "Count" },
          { "Name": "5xx", "Unit": "Count" },
          { "Name": "SuccessLatency", "Unit": "Milliseconds" }
        ],
        "Dimensions": [
          [ "Controller", "Action", "Region", "AZ-ID", "InstanceId"],
          [ "Controller", "Action", "Region", "AZ-ID"],
          [ "Controller", "Action", "Region"]
        ]
      }
    ],
    "LogGroupName": "/loggroupname"
  },
  "CacheRefresh": false,
  "Host": "use1-az2-name.example.com",
  "SourceIp": "34.230.82.196",
  "TraceId": "|e3628548-42e164ee4d1379bf.",
  "Path": "/home",
  "OneBox": false,
  "Controller": "Home",
  "Action": "Index",
  "Region": "us-east-1",
  "AZ-ID": "use1-az2",
  "InstanceId": "i-01ab0b7241214d494",
  "LogGroupName": "/loggroupname",
  "HttpResponseCode": 200,
  "2xx": 1,
  "3xx": 0,
  "4xx": 0,
  "5xx": 0,
  "SuccessLatency": 20
}
```

此日志有三组维度。这三个维度按粒度排序，从实例到可用区再到区域（`Controller` 和 `Action` 始终包含在本示例中）。借助它们，您可以在您的工作负载中创建警报，以指示单个实例、单个可用区或整个 AWS 区域 内的特定控制器操作何时受到影响。这些维度用于记录 2xx、3xx、4xx 和 5xx HTTP 响应计数指标以及成功请求延迟指标（如果请求失败，它还会记录请求失败延迟指标）。该日志还记录其他信息，例如 HTTP 路径、请求者的源 IP 以及此请求是否需要刷新本地缓存。然后，这些数据点可用于计算工作负载提供的每个 API 的可用性和延迟。

关于使用 HTTP 响应代码获取可用性指标的注释

通常，您可以将 2xx 和 3xx 响应视为成功，将 5xx 视为失败。4xx 响应代码介于二者之间，通常是由于客户端错误而生成的。400 响应可能是参数超出范围所致，也可能是所请求内容不存在造成的。您不会将这些响应计入工作负载的可用性范畴，但是，它也可能是软件错误造成的。

例如，如果您引入了更严格的输入验证，拒绝了之前本来可以成功的请求，那么 400 响应可能视作可用性下降。或者，也许是您在限制客户，因此返回 429 响应。虽然限制客户可以保护您的服务，使其维持可用性，但从客户的角度来看，则是您的服务未能处理他们的请求。您需要决定是否将 4xx 响应代码是纳入可用性计算中。

虽然这部分大多使用 CloudWatch 收集和分析指标，但它并不是唯一可以使用的解决方案。您也可以选择将指标发送到 Amazon Managed Service for Prometheus 和 Amazon Managed Grafana（一种 Amazon DynamoDB 表），还可以使用第三方监控解决方案。关键在于，您的工作负载生成的指标必须包含有关工作负载故障隔离边界的上下文。

借助所生成指标的维度与故障隔离边界一致的工作负载，您可以实现检测可用区隔离故障的可观测性。以下部分介绍了三种用于检测因单个可用区受损而导致故障的补充方法。

主题

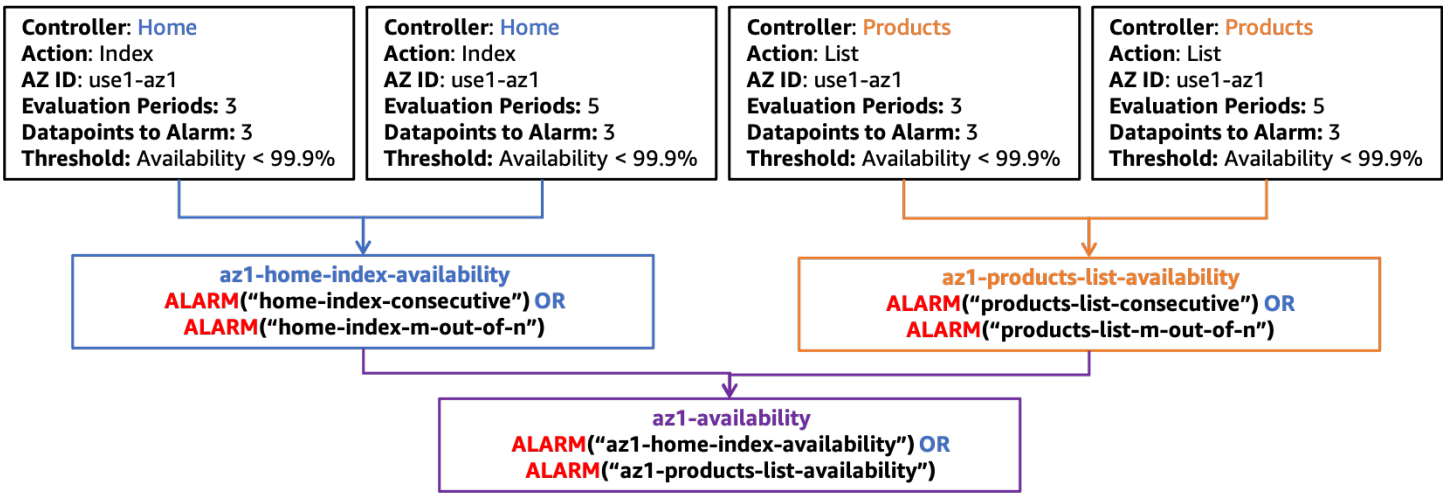
使用 CloudWatch 复合警报进行故障检测

在 CloudWatch 指标中，每个维度集都是一个独特的指标，您可以针对各个维度集创建 CloudWatch 警报。然后，您可以创建 Amazon CloudWatch 复合警报来汇总这些指标。

为了准确检测所产生的影响，本文中的示例将为每个设置了警报的维度集使用两种不同的 CloudWatch 警报结构。每个警报的周期都设置为一分钟，这意味着每分钟对该指标评估一次。第一种方法使用连续的三个超出阈值的数据点，具体来说是将评估期和触发警报的数据点数设置为三，即产生影响的时间总计三分钟。第二种方法是使用“M（最大为 N）”警报，如果将评估期设置为五，将触发警报的数据点数设置为三，表示在五分钟时段内任意 3 个数据点超出阈值。这样可以检测恒定信号以及短时间内波动信号。此处包含的持续时间和数据点数量仅供参考，请根据您的工作负载选择合适的值。

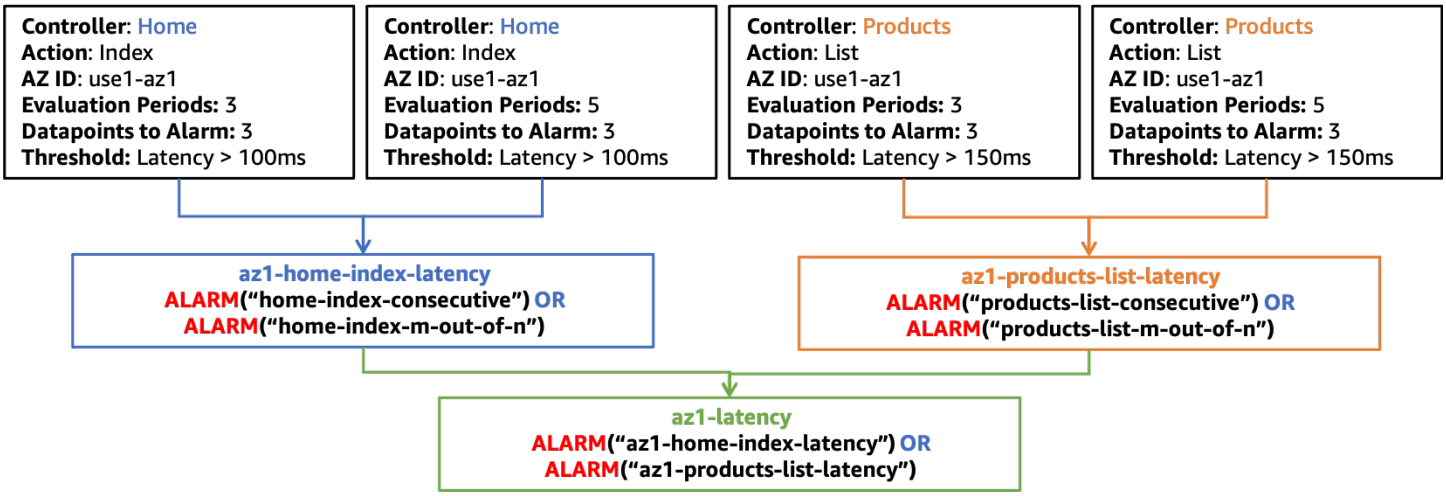
检测在单个可用区中的影响

使用此构造时，可以考虑使用 Controller、Action、InstanceId、AZ-ID 和 Region 维度的工作负载。工作负载有两个控制器，即“产品”和“主页”，每个控制器各有一个操作，分别是“列表”和“索引”。它在 us-east-1 区域的三个可用区中运行。您将为每个可用区中的每个 Controller 和 Action 组合创建两个可用性警报，以及两个延迟警报。然后，您可以选择性地为每个 Controller 和 Action 组合创建可用性复合警报。最后，您创建一个复合警报，该警报汇总了可用区的所有可用性警报。请参见针对单个可用区 use1-az1 的以下图示，该可用区针对各个 Controller 和 Action 组合使用可选的复合警报（use1-az2 和 use1-az3 可用区也具有类似的警报，但为简单起见未显示）。



use1-az1 中可用性复合警报的结构

您还要根据后续图示为延迟构建类似的警报结构。

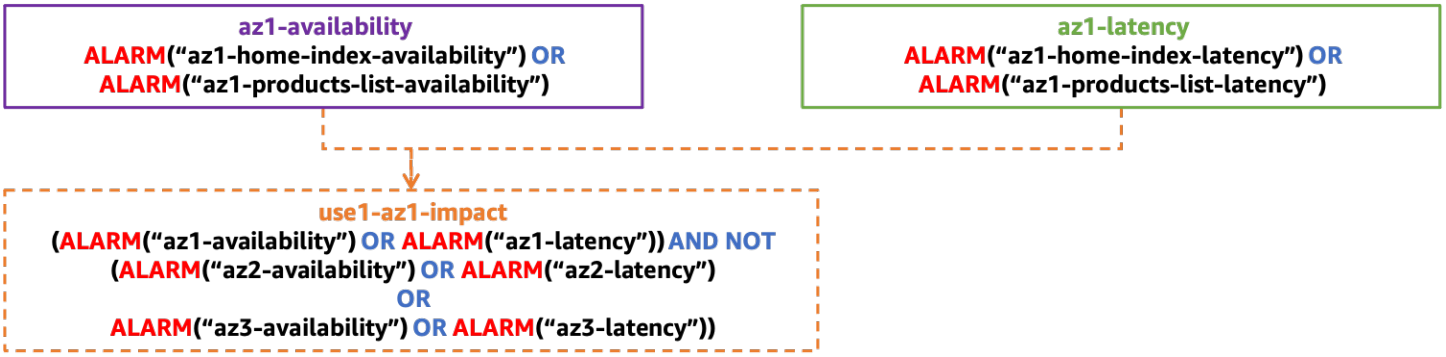


use1-az1 中延迟复合警报的结构

在本部分的其余图示中，只有 az1-availability 和 az1-latency 复合警报将显示在顶层。这些复合警报（az1-availability 和 az1-latency）将告诉您工作负载的任何部分在特定可用区中是否存在可用性低于定义的阈值或延迟超过定义的阈值的情况。您可能还需要考虑测量吞吐量，以检测对您的工作负载在单个可用区中接收工作造成的影响。您也可以将金丝雀发出的指标生成的警报集成到这些复合警报中。这样，如果服务器端或客户端发现可用性或延迟受到影响，就会创建警报。

确保影响不是区域性的

另一组复合警报可用于确保只是隔离的可用区事件激活了警报。实现方法：确保某个可用区的复合警报处于 ALARM 状态时，其他可用区的复合警报仍处于 OK 状态。因此，您使用的每个可用区有一个复合警报。下图显示了一个示例（请注意，use1-az2 和 use1-az3 也有延迟警报和可用性警报：az2-latency、az2-availability、az3-latency 和 az3-availability，为简单起见，未为这些警报绘制图示）。



用于检测对单个隔离的 AZ 的影响的复合警报的结构

确保影响不是由单个实例造成的

单个实例（或整个实例集的一小部分）会对可用性和延迟指标造成影响，严重程度各有差异，看上去对整个可用区都造成影响，而实际上并非如此。与撤离可用区相比，移除单个有问题的实例更快，且同样有效。

实例和容器通常被视为短暂资源，经常被诸如 AWS Auto Scaling 之类的服务所替代。每次创建新实例时都很难创建新的 CloudWatch 警报（当然，使用 Amazon EventBridge 或 Amazon EC2 Auto Scaling 生命周期挂钩还是可能实现的）。但是，您可以使用 CloudWatch Contributor Insights 来确定可用性和延迟指标贡献者的数量。

例如，对于 HTTP Web 应用程序，您可以创建一条规则来识别每个可用区中 5xx HTTP 响应的主要贡献者。这将确定哪些实例导致了可用性下降（我们在上文中定义的可用性指标就是因为存在 5xx 错误的应对举措）。以 EMF 日志为例，使用 InstanceId 的键创建规则。然后，按 HttpResponseCode 字段筛选日志。此示例是 use1-az1 可用区的规则。

```
{
    "AggregateOn": "Count",
    "Contribution": {
        "Filters": [
            {
                "Match": "$.InstanceId",
                "IsPresent": true
            },
            {
                "Match": "$.HttpStatusCode",
                "IsPresent": true
            },
            {
                "Match": "$.HttpStatusCode",
                "GreaterThan": 499
            },
            {
                "Match": "$.HttpStatusCode",
                "LessThan": 600
            },
            {
                "Match": "$.AZ-ID",
                "In": ["use1-az1"]
            },
        ],
        "Keys": [
            "$.InstanceId"
        ]
    },
    "LogFormat": "JSON",
    "LogGroupNames": [
        "/loggroupname"
    ],
    "Schema": {
        "Name": "CloudWatchLogRule",
        "Version": 1
    }
}
```

CloudWatch 警报也可以根据这些规则创建。您可以使用指标数学和带有 `UniqueContributors` 指标的 `INSIGHT_RULE_METRIC` 函数根据 Contributor Insights 规则创建警报。除了可用性指标外，

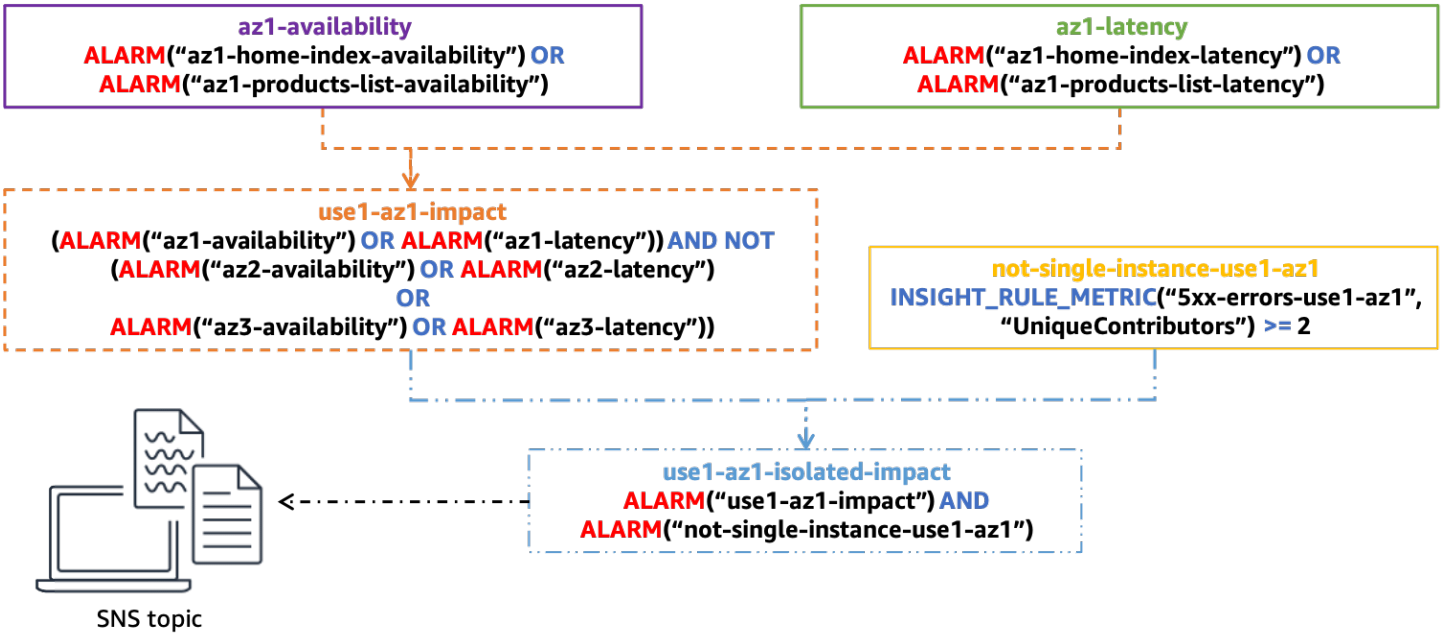
您还可以使用 CloudWatch 警报针对延迟或错误计数等指标创建其他 Contributor Insights 规则。这些警报可以与隔离的可用区影响复合警报一起使用，以确保单个实例不会激活警报。use1-az1 的 Contributor Insights 规则的指标可能会是下面这样：

```
INSIGHT_RULE_METRIC("5xx-errors-use1-az1", "UniqueContributors")
```

当该指标大于阈值（本示例中为 2）时，您可以定义警报。当 5xx 响应的独特贡献者超过该阈值时，表明影响来自两个以上的实例，警报就会被激活。此警报使用大于比较而不是小于比较的原因是为了确保独特贡献者数量为零时不会触发警报。这表示影响不是来自单个实例。您可以根据您的个人工作负载调整此阈值。一般要将该阈值设定为可用区中资源总数的 5% 或更高。如果样本量足够大，超过 5% 的资源受到影响可视为具有统计学显著性。

组合起来

下图显示了单个可用区的完整复合警报结构：



用于确定单可用区影响的完整复合警报结构

当指示隔离可用区延迟或可用性影响的复合警报 use1-az1-aggregate-alarm 处于 ALARM 状态且同一可用区中基于 Contributor Insights 规则的警报 not-single-instance-use1-az1 也处于 ALARM 状态（表示影响来自多个实例）时，会激活最终复合警报 use1-az1-isolated-impact。您将为您的工作负载使用的每个可用区创建此警报堆栈。

您可以将 Amazon Simple Notification Service (Amazon SNS) 提醒附加到最终警报中。之前的所有警报都未配置操作。提醒可以通过电子邮件通知操作员，让其开始手动调查。它还可以启动自动撤离可用

区。但是，请慎重设置对这些提醒的自动响应。撤离可用区后，应该达到以下效果：错误率增加的情况得到缓解，警报恢复到 OK 状态。如果影响发生在另一个可用区，自动化操作可能会撤出第二个或第三个可用区，进而可能移除工作负载的所有可用容量。执行自动化操作前，应检查是否已执行撤离操作。在成功撤离前，您可能还需要扩展其他可用区的资源。

当您向 MVC Web 应用程序添加新的控制器或操作，或者添加新的微服务 - 概括来说就是需要单独监控的任何额外的功能时，您只需修改该设置中的若干警报即可。您将为该新功能创建新的可用性和延迟警报，然后将其添加到相应的可用区对应的可用性和延迟复合警报（在此示例中分别为 `az1-availability` 和 `az1-latency`）。其余的复合警报在配置后保持静态。这样，引入新功能的过程变得更加简单。

使用异常值检测进行故障检测

当多个可用区由于不相关的原因出现错误率上升的情况时，前一种方法的缺陷就显现出来了。假设您在三个可用区部署了 EC2 实例，可用性警报阈值设置为 99%。然后，出现单个可用区损坏，隔离了许多实例，导致该区域的可用性降至 55%。同时，在另一个可用区中，单个 EC2 实例耗尽了其 EBS 卷上的所有存储，因而无法再写入日志文件。这导致它开始返回错误，但它仍可以通过负载均衡器运行状况检查，因为这些检查不会触发写入日志文件。这导致该可用区的可用性降至 98%。这种情况不会激活单个可用区影响警报，因为您面临的可用性影响出现在多个可用区。但是，您仍然可以通过撤离受损的可用区来缓解几乎所有的影响。

在某些类型的工作负载中，您的各种可用区可能会一直出现错误，面对这种情况，之前的可用性指标可能无济于事。以 AWS Lambda 为例。AWS 允许客户创建自己的代码并在 Lambda 函数中运行。要使用该服务，您必须以 ZIP 文件形式上传代码，包括依赖项，并定义函数的入口点。但是客户有时会在一步犯错，例如，他们的 ZIP 文件中遗漏了关键依赖项，或者在 Lambda 函数定义中输入了错误的方法名称。这会造成函数调用失败并导致错误。AWS Lambda 经常遇到这类错误，而这些错误并不表示一定存在运行状况不佳。不幸的是，可用区损坏之类的因素也可能导致这些错误。

要从中发现运行状况不佳造成的错误，您可以使用异常值检测来确定各个可用区的错误数量在统计学上是否存在显著偏差。尽管我们发现多个可用区中出现错误，但如果其中一个可用区确实出现故障，预计该可用区的错误率会比其他可用区高得多，也可能低得多。但是高多少或低多少？

进行此分析的一种方法是使用卡方检测 (χ^2) 来检测可用区之间错误率的统计学显著差异（执行异常值检测有许多不同的算法）。我们来了解一下卡方检验的工作原理。

卡方检验会评估可能出现的结果分布的概率。在本例中，我们关注的是一组确定的可用区中错误的分布。在此示例中，为了简化数学运算，我们使用四个可用区。

首先，建立原假设，定义您认为默认结果会是什么样。在此测试中，原假设是您期望错误在每个可用区中均匀分布。然后，生成备选假设，即错误分布不均匀，则表明可用区受损。现在，您可以使用指标中的数据来检验这些假设。为此，您要对指标进行采样，用时五分钟。假设您在这段时间获得 1000 个已发布的数据点，其中总共有 100 个错误。您预计，如果分布均匀，每个可用区的错误数量占 25%。假设下表是您的预期与实际结果的对比。

表 1：预期错误与实际错误对比

可用区	预期	实际
use1-az1	25	20
use1-az2	25	20
use1-az3	25	25
use1-az4	25	35

您会发现现实中的分布是不均匀的。当然，您可能认为这是由于您采样的数据点存在一定程度的随机性所致。某种程度上，样本集中可能会出现这种类型的分布，仍假设原假设为真。这就引出了以下问题：获得至少如此极端的结果的概率是多少呢？ 如果该概率低于定义的阈值，则拒绝原假设。为具备统计学显著性，该概率应为 5% 或更低。[1]

[1] Craparo, Robert M. (2007)，“显著性级别”，Salkind, Neil J.《测量与统计百科全书》3 卷册，加利福尼亚州千橡市：SAGE Publications，第 889 - 891 页，ISBN 1-412-91611-9。

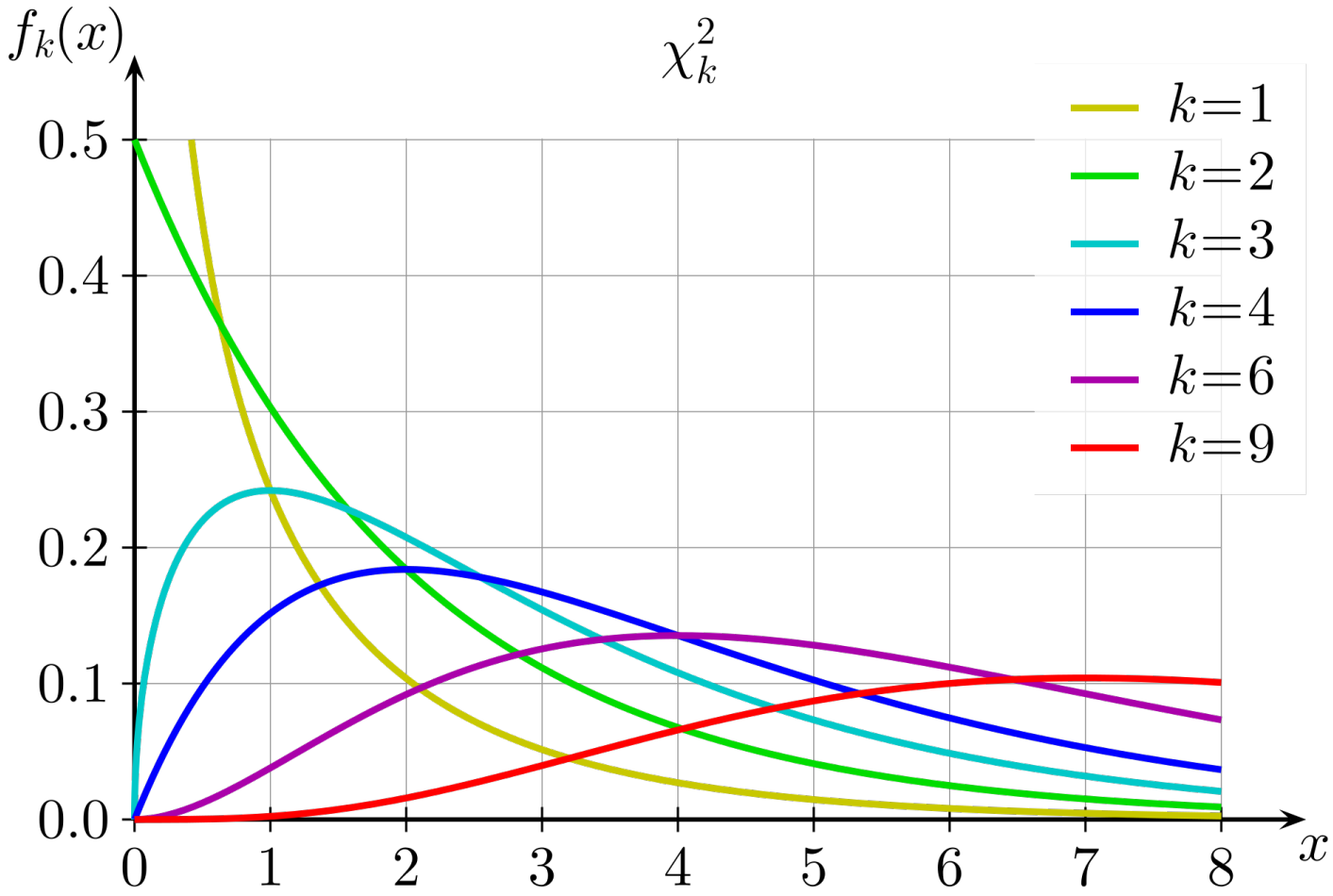
您如何计算这种结果的概率？ 您可以使用 χ^2 统计方法，该统计方法非常适合研究分布，可用于确定使用此公式获得如此极端或更极端结果的概率。

$$E_i = expected\ observations\ of\ type\ i$$
$$O_i = actual\ observations\ of\ type\ i \qquad (1)$$
$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

在我们的示例中，结果是：

$$\chi^2 = \frac{(20-25)^2}{25} + \frac{(20-25)^2}{25} + \frac{(25-25)^2}{25} + \frac{(35-25)^2}{25}$$
$$\chi^2 = \frac{-5^2}{25} + \frac{-5^2}{25} + \frac{0^2}{25} + \frac{10^2}{25} \qquad (2)$$
$$\chi^2 = 1 + 1 + 0 + 4$$
$$\chi^2 = 6$$

在概率方面，6 意味着什么？ 您在查看卡方分布时，需要使用适当的自由度。下图显示了不同自由度下的几个卡方分布。



不同自由度下的卡方分布

自由度的计算方法是用检验中的选择数减一。在本示例中，由于有四个可用区，因此自由度为三。然后，您需要求出 k = 3 且 x ≥ 6 的曲线下方的面积（积分）。您也可以使用预先计算的表，其中包含接近该值的常用值。

表 2：卡方检验临界值表

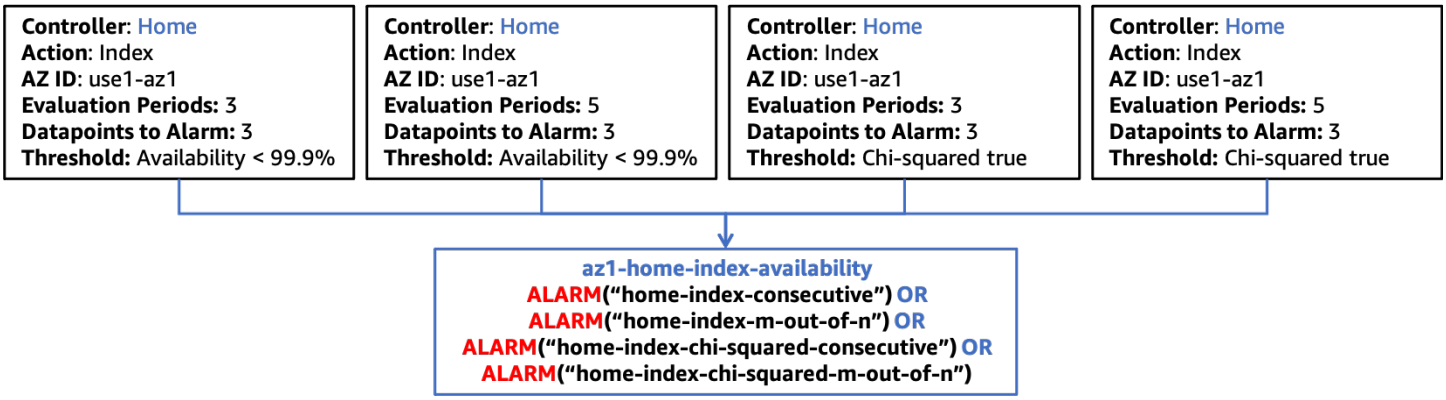
自由度	小于临界值的概率				
	0.75	0.90	0.95	0.99	0.999
1	1.323	2.706	3.841	6.635	10.828
2	2.773	4.605	5.991	9.210	13.816
3	4.108	6.251	7.815	11.345	16.266
4	5.385	7.779	9.488	13.277	18.467

自由度为 3 时，卡方值 6 位于 0.75 和 0.9 概率列之间。这意味着出现这种分布的可能性超过 10%，并不低于阈值 5%。因此，您接受原假设并确定可用区中的错误率不具备统计学显著差异。

CloudWatch 指标数学本身不支持执行卡方统计检验，因此您需要从 CloudWatch 收集适用的错误指标，然后在 Lambda 等计算环境中运行测试。您可以决定在 MVC 控制器/操作、单个微服务级别，还是在可用区级别执行此测试。您需要考虑可用区损坏是否会对每个控制器/操作或微服务产生同等影响，或者诸如 DNS 故障之类的事情是否会对低吞吐量服务造成影响，而不会对高吞吐量服务造成影响，因为高吞吐量服务可能会在聚合时掩盖影响。无论哪种情况，都要选择相应的维度来创建查询。粒度级别也会影响您创建的 CloudWatch 警报。

收集指定时间内每个可用区和控制器/操作的错误计数指标。首先，将卡方检验的计算结果设置为 true（存在统计学显著偏差）或 false（不存在统计学显著偏差，也就是说，原假设成立）。如果结果为 false，则向指标流发布一个 0 数据点作为每个可用区的卡方检验结果。如果结果为 true，则为错误与预期值相差最远的可用区发布一个 1 数据点，为其他可用区发送一个 0 数据点（有关可用于 Lambda 函数的示例代码，请参阅 附录 B – 卡方计算示例）。您可以采用与之前的可用性警报相同的方法，即根据 Lambda 函数生成的数据点创建连续三个 CloudWatch 指标警报和“3（最大为 5）”CloudWatch 指标警报。与之前的示例一样，您可以修改其中的数据，以缩短或延长时间，增加或减少数据点。

然后，将这些警报添加到控制器和操作组合现有的可用区可用性警报中，如下图所示。



将卡方统计检验与复合警报相集成

如前所述，当您在工作负载中加入新功能时，您只需相应地创建针对该新功能的 CloudWatch 指标警报，然后更新复合警报层次结构中的下一层以包含这些警报。警报结构的其余部分保持静态。

对单实例区域资源进行故障检测

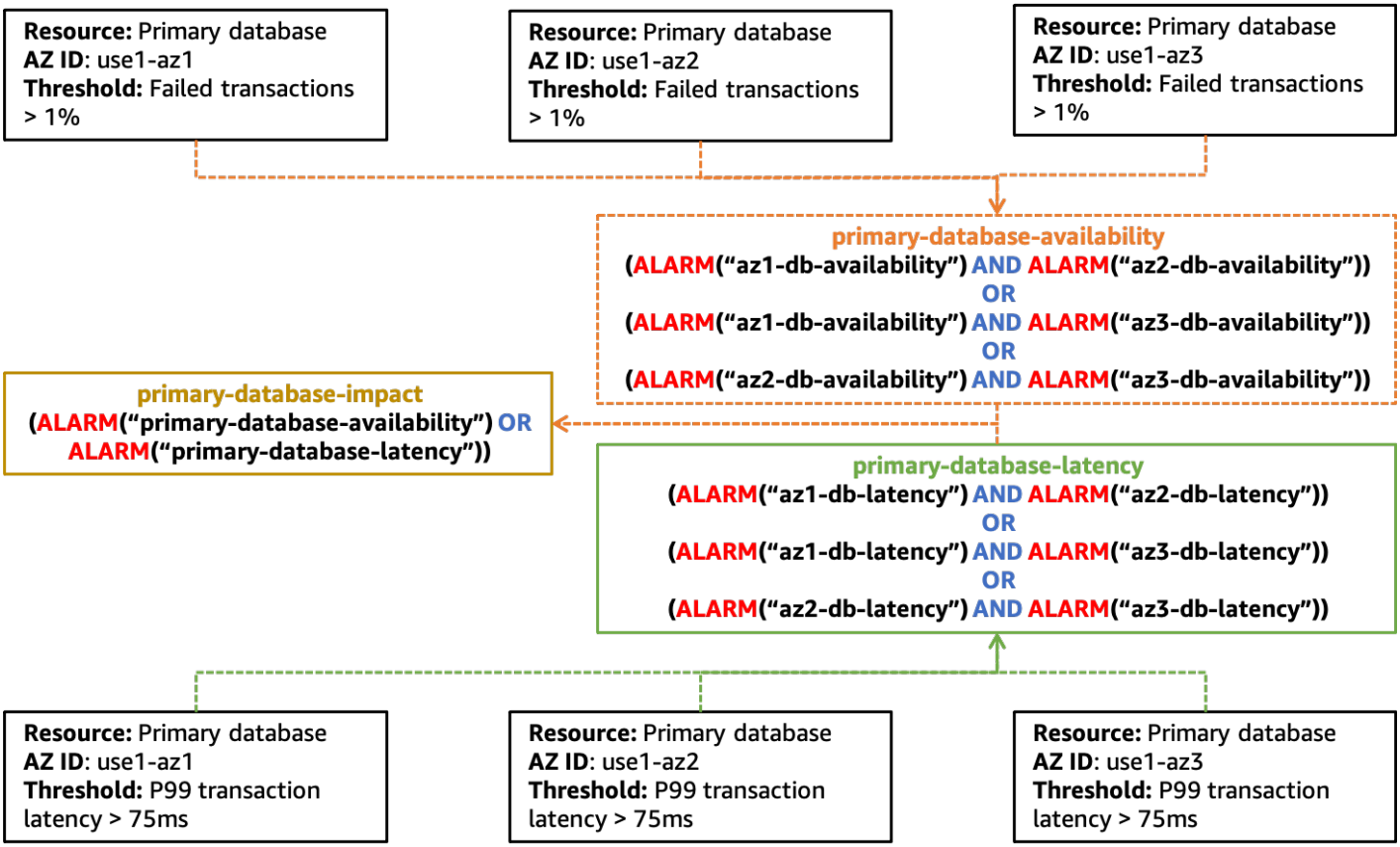
在某些情况下，您可能拥有区域资源的单个活动实例，最常见的是需要单写入器组件的系统，例如关系数据库（例如 Amazon RDS）或分布式缓存（例如 Amazon ElastiCache for Redis）。如果单个可用区损坏影响了主资源所在的可用区，则可能会影响访问该资源的每个可用区。这可能会导致每个可用区都超出可用性阈值，如果出现这种情况，第一种方法便无法正确识别单个可用区的影响来源。此外，还可能出现每个可用区错误率相似的情况，这意味着异常值分析也无法检测到问题。这意味着您需要实现额外的可观测性来专门检测这种情况。

您担心的资源很可能会生成自己的运行状况指标，但是在可用区受损期间，该资源可能无法提供这些指标。在这种情况下，您应该创建或更新警报，以了解自己何时处于盲目状态。如果您已经监控了重要指标并对其设置了警报，则可以将警报配置为将缺失数据视为违例。这有助于您了解资源是否停止报告数据，并且可以包含在以前使用的连续和 M（最大为 N）警报中。

在某些表明资源运行状况的指标中，也有可能在没有活动时发布一个值为零的数据点。如果损坏情况阻碍了与资源的交互，那么您无法对这类指标使用缺失数据方法。您可能也不希望在值为零时发出警报，因为在某些合法场景中，该值可能在正常阈值之内。检测此类问题的最佳方法是使用此依赖项由资源生成指标。在本例中，我们希望使用复合警报来检测多个可用区域的影响。这些警报应使用与资源相关的几个关键指标类别。下面列出了几个例子：

- 吞吐量 – 传入工作单元的速率。这可能是事务、读取、写入等。
- 可用性 – 衡量成功与失败的工作单元的数量。
- 延迟 – 测量跨关键操作成功执行工作的延迟的多个百分位数。

同样，您可以为要衡量的每个指标类别中的每个指标创建连续和 M（最大为 N）指标警报。和以前一样，这些警报可以合并成一个复合警报，以确定此共享资源是各个可用区的影响来源。您希望能够使用复合警报发现对多个可用区的影响，但影响不一定波及所有可用区。这种方法的复合警报的高级结构如下图所示。



创建警报来检测单个资源对多个可用区造成影响的示例

您会注意到，此图表对应使用哪种类型的指标警报以及复合警报的层次结构的规定较少。这是因为发现此类问题可能很困难，并且需要仔细注意共享资源的正确信号。可能还需要以特定的方式评估这些信号。

此外，您应该注意到该 `primary-database-impact` 警报并未与特定可用区关联。这是因为您可以将主数据库实例配置为使用任意可用区，因此它可能存在于任意可用区，并且没有 CloudWatch 指标指定其所在位置。当您看到此警报已激活时，应将其用作资源可能存在问题的信号，并启动失效转移到另一个可用区（如果尚未自动执行）。将资源转移到另一个可用区后，您可以稍等一下，看看隔离的可用区警报是否已激活，也可以选择先发制人，调用您的可用区撤离计划。

摘要

本部分介绍了三种帮助发现单个可用区损坏的方法。您应将三种方法结合使用，以全面了解工作负载的运行状况。

CloudWatch 复合警报方法可以发现可用性偏差在统计学上不显著的问题，例如 98%（受损的可用区）、100% 和 99.99% 的可用性，这不是由单个共享资源引起的。

异常值检测将帮助检测单个可用区损坏，即多个可用区中发生不相关的错误，且这些错误都超过了您的警报阈值。

最后，发现单个实例区域资源的降级有助于发现可用区损坏何时会影响跨可用区共享的资源。

您可以将每种模式产生的警报组合成 CloudWatch 复合警报层次结构，以发现单个可用区何时发生损坏，以及何时会影响工作负载的可用性或延迟。

可用区撤离模式

检测到受影响的单个可用区后，下一步是撤离该可用区。疏散需要达到两个效果。

首先，您想要停止向受影响的可用区发送工作。在不同的架构中，具体的工作也会不同。在请求/响应工作负载中，这意味着停止将来自客户的 HTTP 或 gRPC 请求发送到该可用区的负载均衡器或其他资源。在成批处理或队列处理系统中，这可能意味着停止计算资源处理受影响可用区中的工作。您还需要防止未受影响的可用区中的资源与受影响可用区中的资源交互，例如，EC2 实例向受影响可用区内的接口 VPC 端点发送流量，或连接到数据库的主实例。

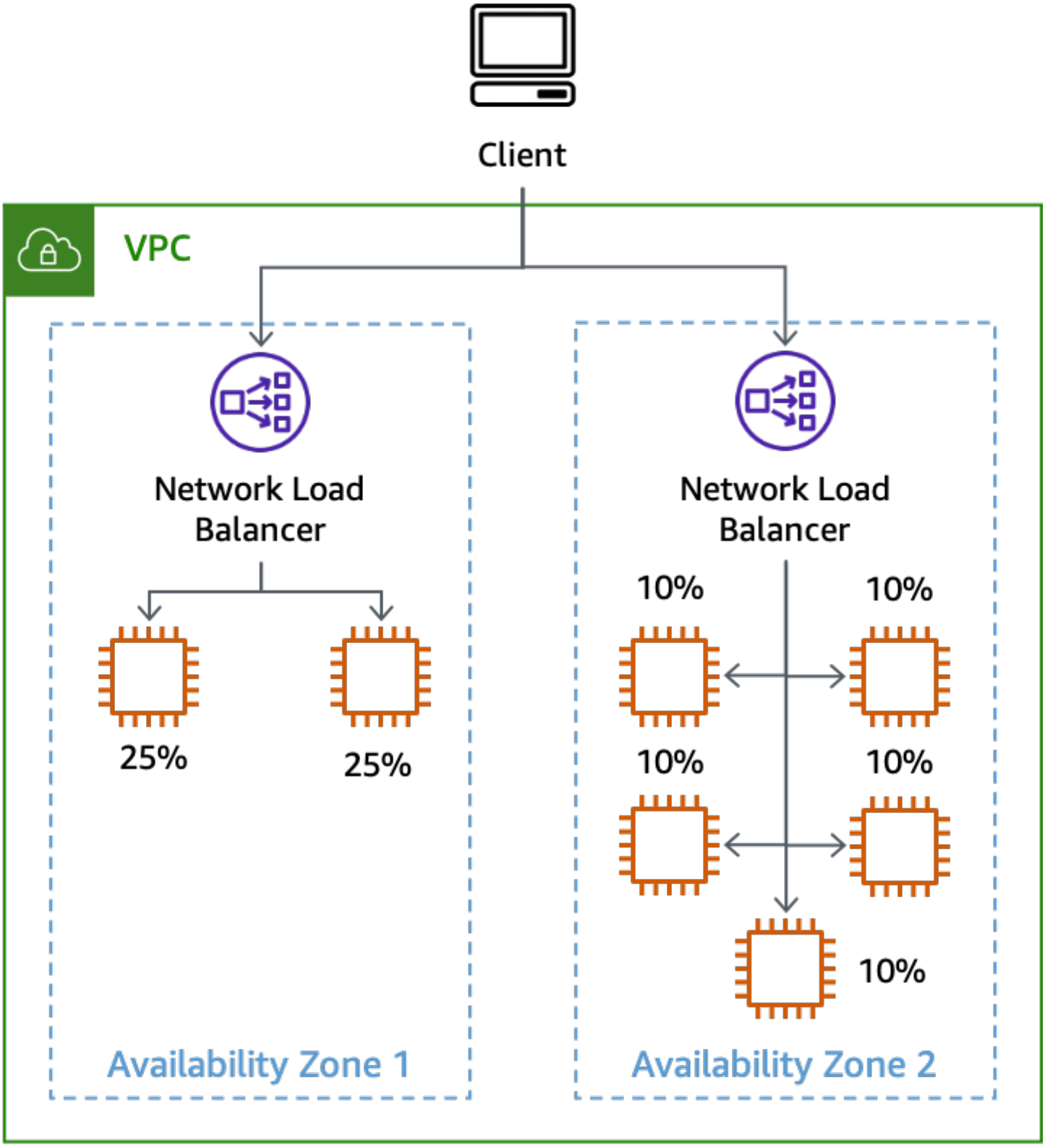
第二个效果是阻止在受影响可用区内配置新容量。这一点很重要，因为在受影响可用区中配置的新资源（如 EC2 实例或容器）可能会受到与现有资源相同的影响。此外，由于第一个效果会阻止向它们发送工作，因此他们无法接收到本应配置给它们处理的负载。这会导致现有资源的负荷增加，最终导致工作负载减少或完全丢失。在适用情况下，AWS 可提供几种自动扩展服务：Amazon EC2 Auto Scaling、Application Auto Scaling 和 AWS Auto Scaling。此外，Amazon ECS、Amazon EKS 和 AWS Batch 可以在 VPC 中跨可用区为主机安排工作，这是其正常运行的一部分。

主题

可用区独立性

要实现第一个效果，即停止向受影响的可用区发送工作，您需要实现可用区独立性 (AZI)，有时也称可用区亲和性，来实现撤离。这种架构模式隔离了可用区内的资源，并防止不同可用区中的资源之间进行交互，除非绝对需要（如连接到其他可用区中的主数据库实例）。

在请求/响应类型的工作负载中，要实现 AZI，您需要禁用应用程序负载均衡器 (ALB)、经典负载均衡器 (CLB) 和网络负载均衡器 (NLB) 的跨区域负载均衡（默认情况下，NLB 的跨区域负载均衡处于禁用状态）。禁用跨区域负载均衡时，需要做出权衡。禁用跨区域负载均衡后，无论各个可用区中有多少实例，流量都会平均分配给每个可用区。如果您有不均衡的资源或自动扩缩组，则可能会给资源较少的可用区造成额外负担。如下图所示，可用区 1 中有两个实例，各接收 25% 的负载，而可用区 2 中有五个实例，各接收 10% 的负载。



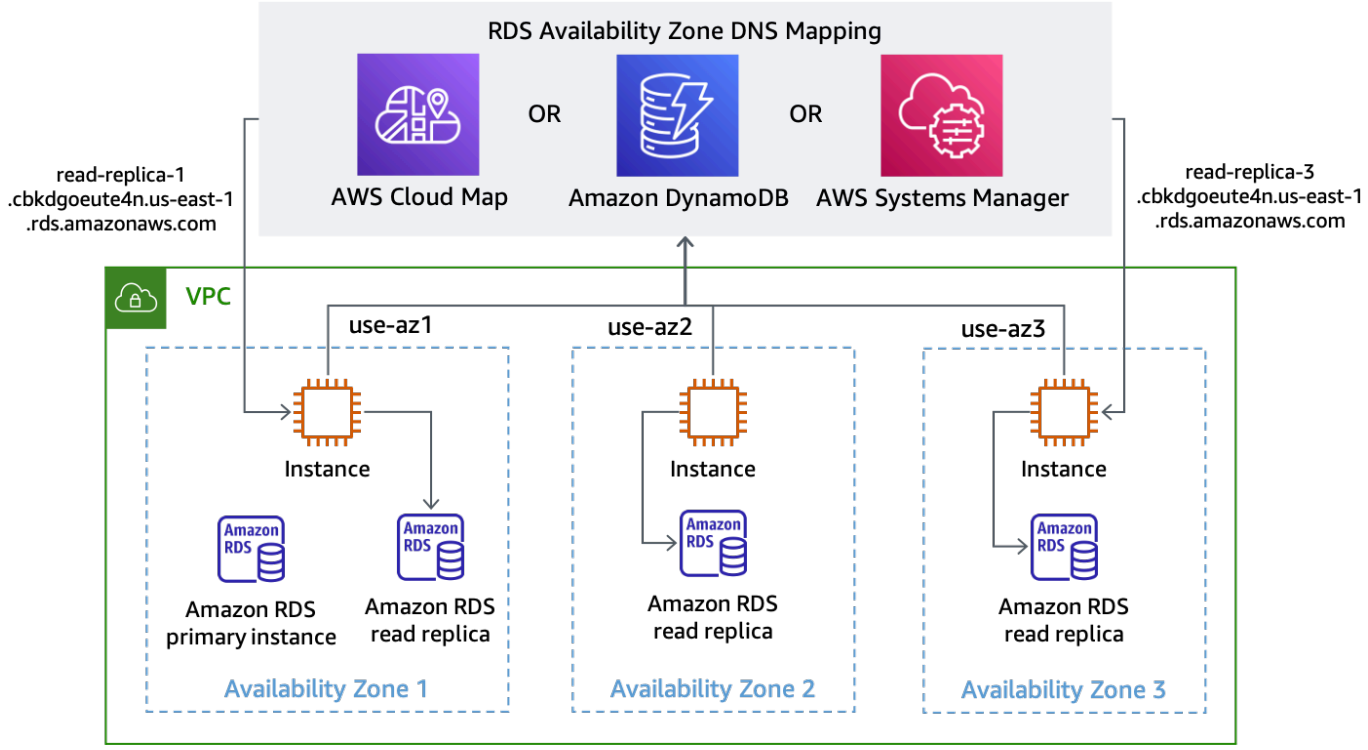
实例不均衡时禁用跨区域负载均衡的效果

您使用的其他区域服务也需要实现 AZI 模式，以有效撤离可用区。例如，接口 VPC 端点为接口端点所在的每个可用区提供特定的 DNS 名称。

实现 AZI 的一个挑战在于数据库，尤其是大多数关系数据库在任何时候都只支持单个主写入器。与主实例通信时，您可能需要跨越可用区边界。许多 AWS 数据库服务都支持用户定义的多可用区配置，并具有内置的多可用区失效转移功能，例如 Amazon RDS 或 Amazon Aurora。在许多故障场景中，该服务可以检测到影响，并在出现问题时自动将数据库失效转移到其他可用区。但是，在灰色故障中，服务可能无法检测到其对您的工作负载的影响，或者不认为此影响与数据库相关。在这些情况下，如果您检测到可用区受到影响，可以手动调用失效转移来移动主数据库。这样能够有效地应对单个可用区受损情况。

如果您在这些数据库中使用只读副本功能，则可能还需要为这些数据库实现 AZI，因为您无法像处理主数据库那样将只读副本失效转移到其他可用区。如果您在可用区 1 中有一个只读副本，并且将三个可用区的实例都配置为使用该副本，则影响可用区 1 的损坏也会影响其他两个可用区的运行。您需要防止出现这种影响。

对于 RDS 实例，您将收到一个 DNS 端点，用于访问特定可用区中的副本。要实现 AZI，您需要为每个可用区提供一个只读副本，并让您的应用程序了解其所在可用区使用哪个副本端点。要实现这种操作，一种方法是将可用区 ID 附加到数据库标识符中，比如 `use1-az1-read-replica.cbkdgoeute4n.us-east-1.rds.amazonaws.com`。另一种方法是使用服务发现（例如通过 AWS Cloud Map）或查找存储在 AWS Systems Manager Parameter Store 或 DynamoDB 表中的简易地图。请参见下图，了解此概念。

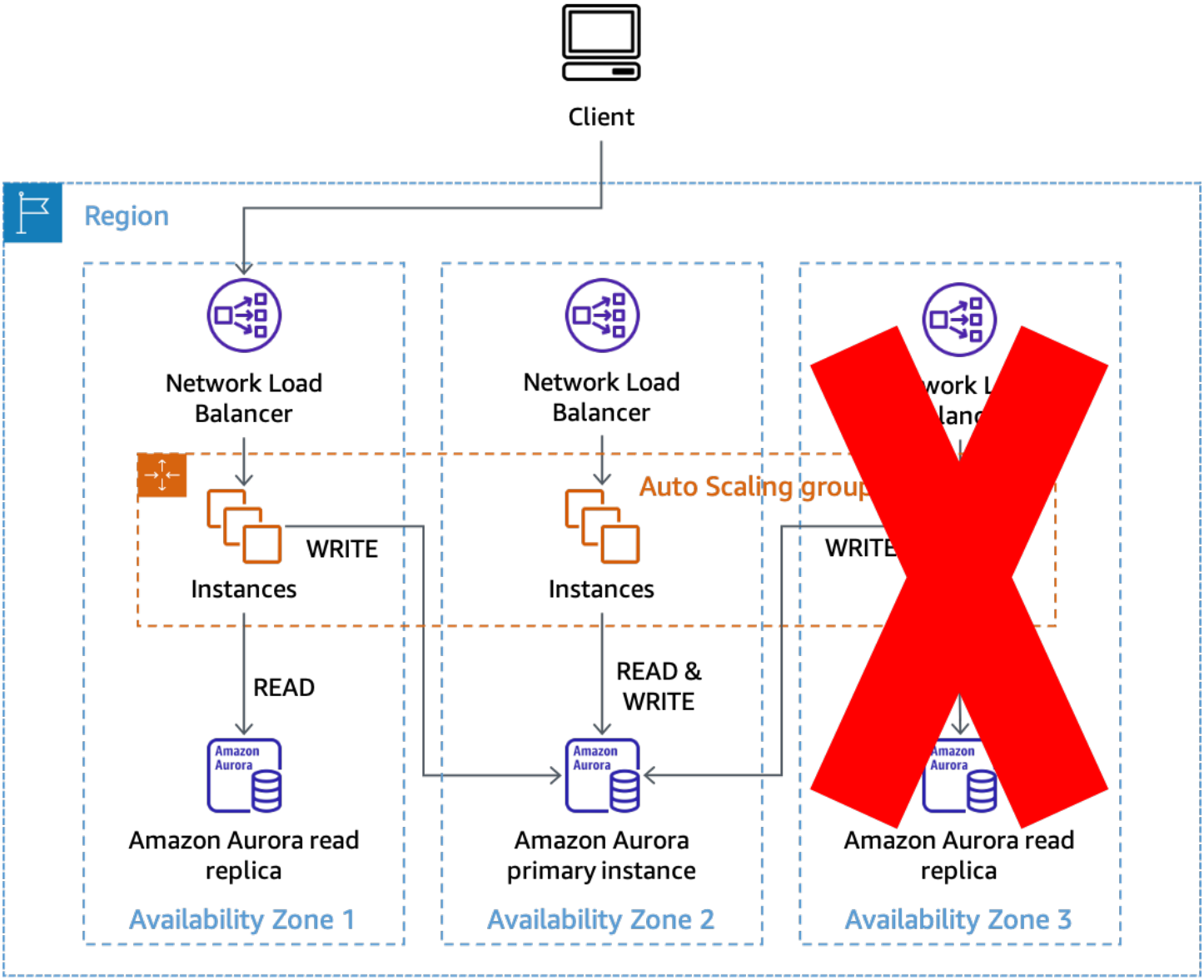


查找每个可用区的 RDS 端点 DNS 名称

Amazon Aurora 的默认配置是提供单个读取器端点，用于在可用只读副本之间对请求进行负载均衡。要使用 Aurora 实现 AZI，您可以为每个使用 ANY 类型的只读副本使用自定义端点（这样您就可以在需要时提升只读副本）。根据部署该副本的可用区 ID 命名自定义端点。然后，您可以使用自定义端点提供的 DNS 名称连接到特定可用区中的特定只读副本，如下图所示。

为 Aurora 只读副本使用自定义端点

如果您的系统采用这种架构，撤离可用区就简单多了。例如，在下图中，可用区 3 受到损坏的影响，而可用区 1 和 2 中的读取和写入操作并不会受到影响。



使用 Amazon Aurora 只读副本实现 AZI 来防止受到影响

换一种情况，如果可用区 2 受到影响，可用区 1 和 3 中的读取操作仍可以成功执行。此时如果 Amazon Aurora 没有自动对主数据库进行失效转移，您可以手动将其失效转移到其他可用区，以恢复处理写入的能力。借助这种方法，您在撤离可用区时，无需对数据库连接进行任何配置更改。需要进行的更改越少，流程越简单，则操作越可靠。

控制面板和数据面板

在了解撤离可用区时所使用的具体模式前，我们需要了解一下控制面板和数据面板的概念。在我们的服务中，AWS 对控制面板和数据面板做出了明确区分。控制面板是对系统进行更改（添加资源、删除资

源、修改资源）以及将这些更改传播到任何所需位置（例如更新 ALB 的网络配置或创建 AWS Lambda 函数）所涉及的机器。

数据面板是这些资源的主要功能，例如运行 EC2 实例，或者在 Amazon DynamoDB 表中存取项目。有关控制面板和数据面板的更多信息，请参阅使用可用区的静态稳定性和 AWS 故障隔离边界。

就本文档而言，控制面板的活动部件和依赖关系比数据面板要多。从统计学上讲，控制面板比数据面板受损的可能性更大。这对于提供 AZI 的服务（例如 Amazon EC2 和 EBS）尤为重要，因为其中部分服务的控制面板还不受区域限制，会在单可用区事件中受到损坏。

虽然控制面板操作可用于执行可用区撤离，但根据上述信息，这些操作的成功率可能较低，尤其是在发生故障事件期间。为了提高缓解影响的成功率，可以使用两种不同的模式。第一种模式仅依靠数据面板操作，通过阻止将工作路由到受影响的可用区或停止在受影响的可用区中执行工作来缓解影响。而在第二种模式中，我们会通过控制面板操作更新资源配置，从而防止在受影响的可用区中配置容量，并停止受影响的可用区与其他可用区之间的通信。

本部分要讨论的恢复模式是紧急应对措施。这种模式的机制就是快速采取大规模措施，类似于消防站响起警铃。采用该模式，即表示已经对工作负载的代码尝试了采用指数回退和抖动的重试等策略，以克服暂时性错误。这意味着，当检测到隔离的可用区的影响时，其对可用性或延迟的影响非常严重，需要撤离该可用区才能有效缓解影响。

通过数据面板控制的撤离

多种解决方案支持仅使用数据面板即可撤离可用区的操作。本部分将介绍其中三种解决方案以及使用案例，您可以根据需要进行选择。

使用任一这些解决方案时，您都需要确保剩余的可用区具备足够的容量来处理要撤离的可用区中的负载。要做到这一点，最有弹性的方法是为每个可用区预先配置所需的容量。如果您正在使用三个可用区，则除了所需容量外，每个可用区中额外部署 50% 的容量来处理峰值负载。这样，如果失去一个可用区，您所拥有的容量可以 100% 满足需求，而不必使用控制面板来配置更多容量。

此外，如果您正在使用 EC2 自动扩缩服务，请确保您的自动扩缩组 (ASG) 在转移期间不会横向缩减，这样当转移结束时，您的组中仍有足够的容量来处理您的客户流量。这一点，需要您确保自动扩缩组 (ASG) 的最低所需容量能够处理您当前的客户负载。您还可以通过在指标中使用平均值，而不是异常百分位数指标（如 P90 或 P99）来确保自动扩缩组 (ASG) 不会意外缩小。

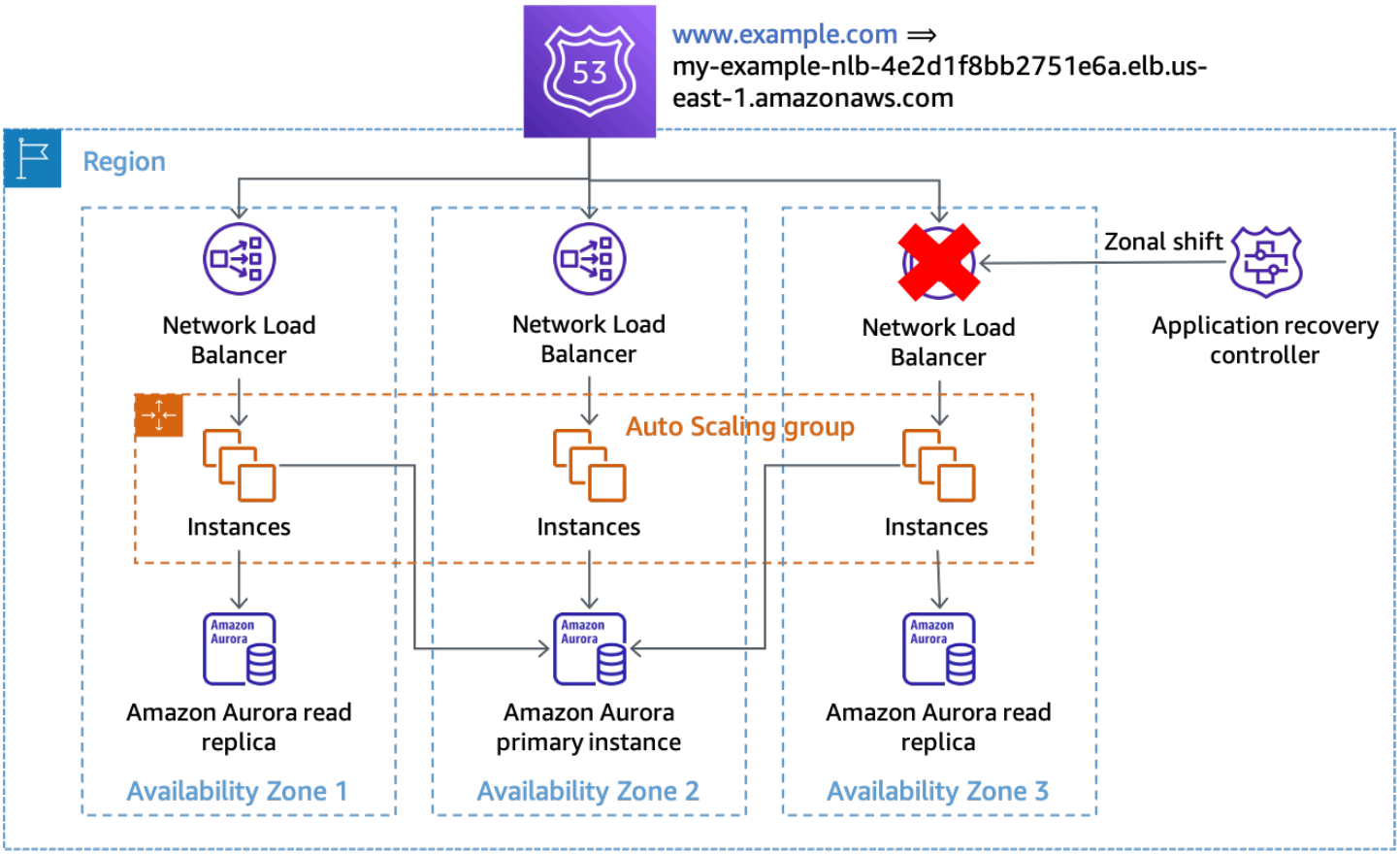
在转移过程中，不再处理流量的资源的利用率应该非常低，而其他资源的利用率会因处理新流量而出现提升。此时要保持平均值处于稳定状态，这样可以防止出现横向缩减操作。最后，您还可以对 ALB 和

NLB 进行目标组运行状况设置，根据健康主机的百分比或数量来指定 DNS 故障转移。这样可以防止流量路由到没有足够正常运行主机的可用区。

Route 53 应用程序恢复控制器 (ARC) 中的可用区转移

第一个可用区撤离解决方案使用 Route 53 ARC 中的可用区转移。此解决方案适用于将 NLB 或 ALB 作为客户流量入口点的请求/响应工作负载。

当您检测到可用区受损时，可以使用 Route 53 ARC 启动可用区转移。此操作完成且现有缓存的 DNS 响应过期后，所有新请求将仅路由到剩余可用区中的资源。下图显示了可用区转移的工作原理。在下图中，我们有一条 `www.example.com` 的Route 53 别名记录，指向 `my-example-nlb-4e2d1f8bb2751e6a.elb.us-east-1.amazonaws.com`。可用区转移是对可用区 3 执行的。



可用区转移

在示例中，如果主数据库实例不在可用区 3 中，则只需执行可用区转移即可实现撤离可用区的第一个效果，即阻止在受影响的可用区中处理工作。如果主节点位于可用区 3 中，且 Amazon RDS 未自动进行失效转移，那么您可以配合可用区转移手动启动失效转移（这需要使用 Amazon RDS 控制面板）。该操作适用于本部分介绍的所有通过数据面板控制的解决方案。

您应该使用 CLI 命令或 API 启动可用区转移，以最大限度地减少开始撤离所需的依赖项。撤离过程越简单，就越可靠。具体命令可以存储在本地运行手册中，以便待命工程师使用。可用区转移是撤离可用区时最推荐、最简单的解决方案。

Route 53 ARC

第二种解决方案使用 Route 53 ARC 的功能来手动指定特定 DNS 记录的运行状况。该解决方案的优点是使用了高度可用的 Route 53 ARC 集群数据面板，能够应对两个 AWS 区域 受损的状况。但是该解决方案会产生额外的成本，并且需要额外配置 DNS 记录。要实现此模式，您需要为负载均衡器（ALB 或 NLB）提供的可用区特定 DNS 名称创建别名记录。如下表所示：

表 3：为负载均衡器的区域 DNS 名称配置的 Route 53 别名记录

路由策略：加权	路由策略：加权	路由策略：加权
名称：www.example.com	名称：www.example.com	名称：www.example.com
类型：A（别名）	类型：A（别名）	类型：A（别名）
值：us-east-1b.load-balancer-name.elb.us-east-1.amazonaws.com	值： us-east-1a.load-balancer-name.elb.us-east-1.amazonaws.com	值： us-east-1c.load-balancer-name.elb.us-east-1.amazonaws.com
权重：100	权重：100	权重：100
评估目标运行状况：true	评估目标运行状况：true	评估目标运行状况：true

对于每个 DNS 记录，您需要配置与 Route 53 ARC 路由控制关联的 Route 53 运行状况检查。如果您要启动可用区撤离，请将路由控制状态设置为 Off。AWS 建议您使用 CLI 或 API 执行此操作，以最大限度地减少开始可用区撤离所需的依赖项。执行此操作时的最佳实操是在本地保留 Route 53 ARC 集群端点的副本，这样当需要撤离时，就无需从 ARC 控制面板中检索这些端点。

为了最大限度地降低使用此方法时的成本，您可以在单个 AWS 账户 中创建单个 Route 53 ARC 集群和运行状况检查，并与组织中的其他 AWS 账户 共享运行状况检查。采用这种方法时，您应使用可用区 ID (AZ-ID)（例如，use1-az1），而非可用区名称（例如，us-east-1a）来进行路由控制。这是因为 AWS 会将物理可用区随机映射到各个 AWS 账户 的可用区名称，而使用 AZ-ID 则可以始终如一地引用相同的物理位置。当您启动可用区撤离时，比如说 use1-az2，应确保每个 AWS 账户 中的 Route 53 记录集都使用 AZ-ID 映射为各个 NLB 记录配置正确的运行状况检查。

例如，假设某个 Route 53 运行状况检查与 use1-az2 的 Route 53 ARC 路由控制相关联，ID 为 0385ed2d-d65c-4f63-a19b-2412a31ef431。如果其他 AWS 账户 想要使用此运行状况检查，且将 us-east-1c 映射到 use1-az2，那么针对记录 us-east-1c.load-balancer-name.elb.us-east-1.amazonaws.com，您要使用 use1-az2 运行状况检查。在资源记录集中，您要使用运行状况检查 ID 0385ed2d-d65c-4f63-a19b-2412a31ef431。

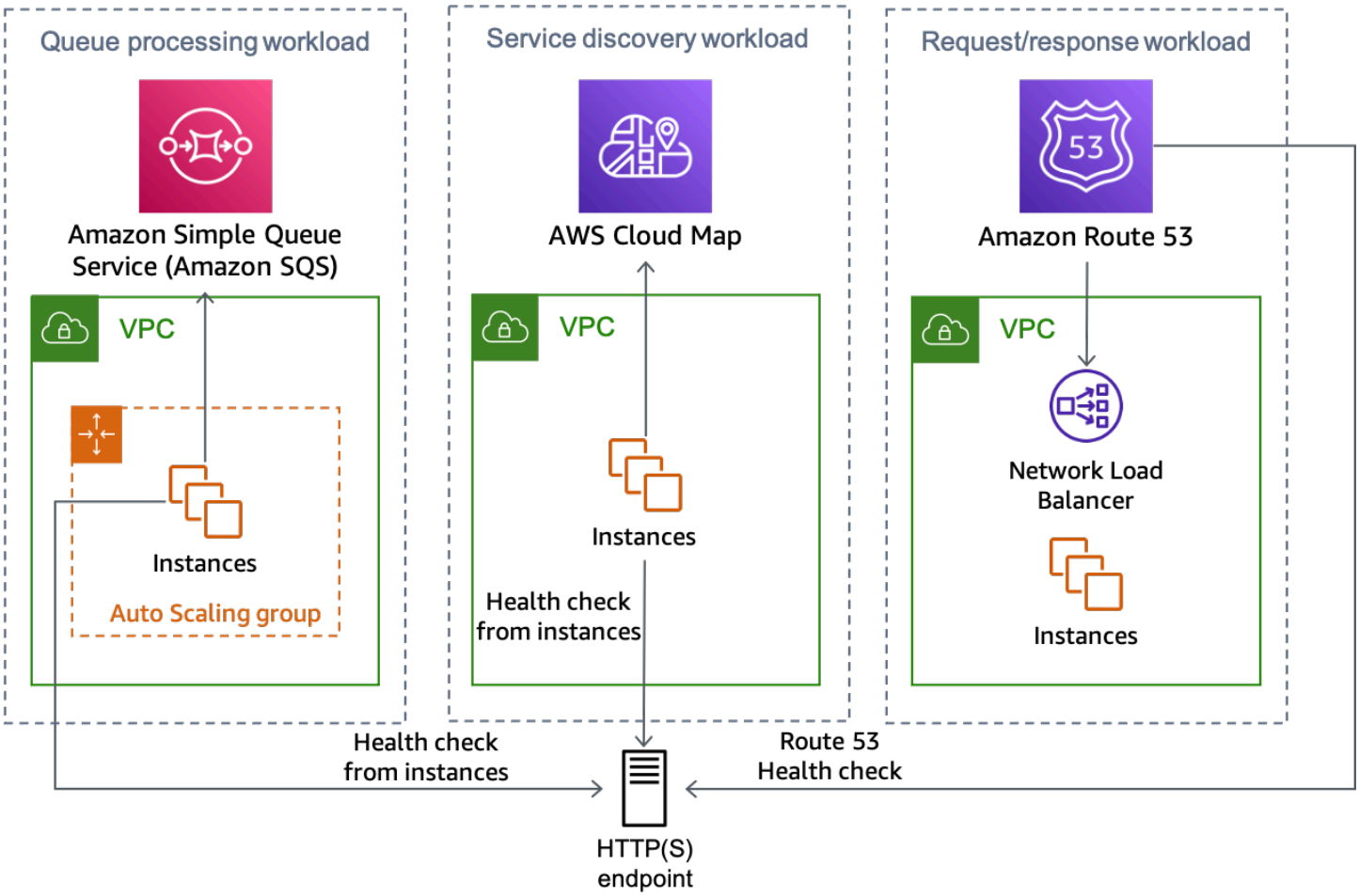
使用自托管 HTTP 端点

您也可以通过管理自己的 HTTP 端点（用于指示特定可用区的状态）来实现此解决方案。它允许您根据来自 HTTP 端点的响应手动指定可用区何时运行状况不佳。该解决方案的成本低于使用 Route 53 ARC，但高于可用区转移，并且需要管理额外的基础设施。它的优势是可以更灵活地应对不同的场景。

该模式可以搭配使用 NLB 或 ALB 架构以及 Route 53 运行状况检查。它也可以用于非负载均衡的架构，例如服务发现或队列处理系统，在这些系统中，Worker 节点自行执行运行状况检查。在这些场景中，主机可以使用后台线程，使用其 AZ-ID（有关如何查找该 ID 的信息，请参阅 附录 A – 获取可用区 ID ）定期向 HTTP 端点发出请求，然后接收有关可用区运行状况的响应。

如果可用区运行状况不佳，他们有多种响应方式可供选择。他们可以选择使来自 ELB、Route 53 等来源的外部运行状况检查或服务发现架构中的自定义运行状况检查失败，使其在这些服务中显示为状况不佳。他们也可以在收到请求后立即发出存在错误的响应，让客户端退避并重试。在事件驱动的架构中，节点可以故意无法处理工作，例如故意向队列返回 SQS 消息。在工作路由器架构（中央服务安排特定主机的工作）中，您也可以使用这种模式。路由器可以在选择工作线程、端点或单元之前检查可用区的状态。在使用 AWS Cloud Map 的服务发现架构中，您可以通过在请求中提供筛选条件（例如 AZ-ID）来发现端点。

下图显示了如何将这种方法用于多种类型的工作负载。



多种工作负载类型均可使用 HTTP 端点解决方案

您可以通过多种方法实现 HTTP 端点，接下来将简要介绍其中两种。

使用 Amazon S3

这种模式最初出现在该博客文章中，用于多区域灾难恢复。您可以将相同的模式用于可用区撤离。

在这种场景中，您将为每个区域 DNS 记录创建 Route 53 DNS 资源记录集（与上面的 Route 53 ARC 场景相似）和相关的运行状况检查。但是，在此实施中，您无需将运行状况检查与 Route 53 ARC 路由控制相关联，而是将其配置为使用 HTTP 端点，并进行反转以防止 Amazon S3 中的损坏意外触发撤离。当对象不存在时，运行状况检查结果为正常；当对象存在时，结果为不正常。设置如下表所示。

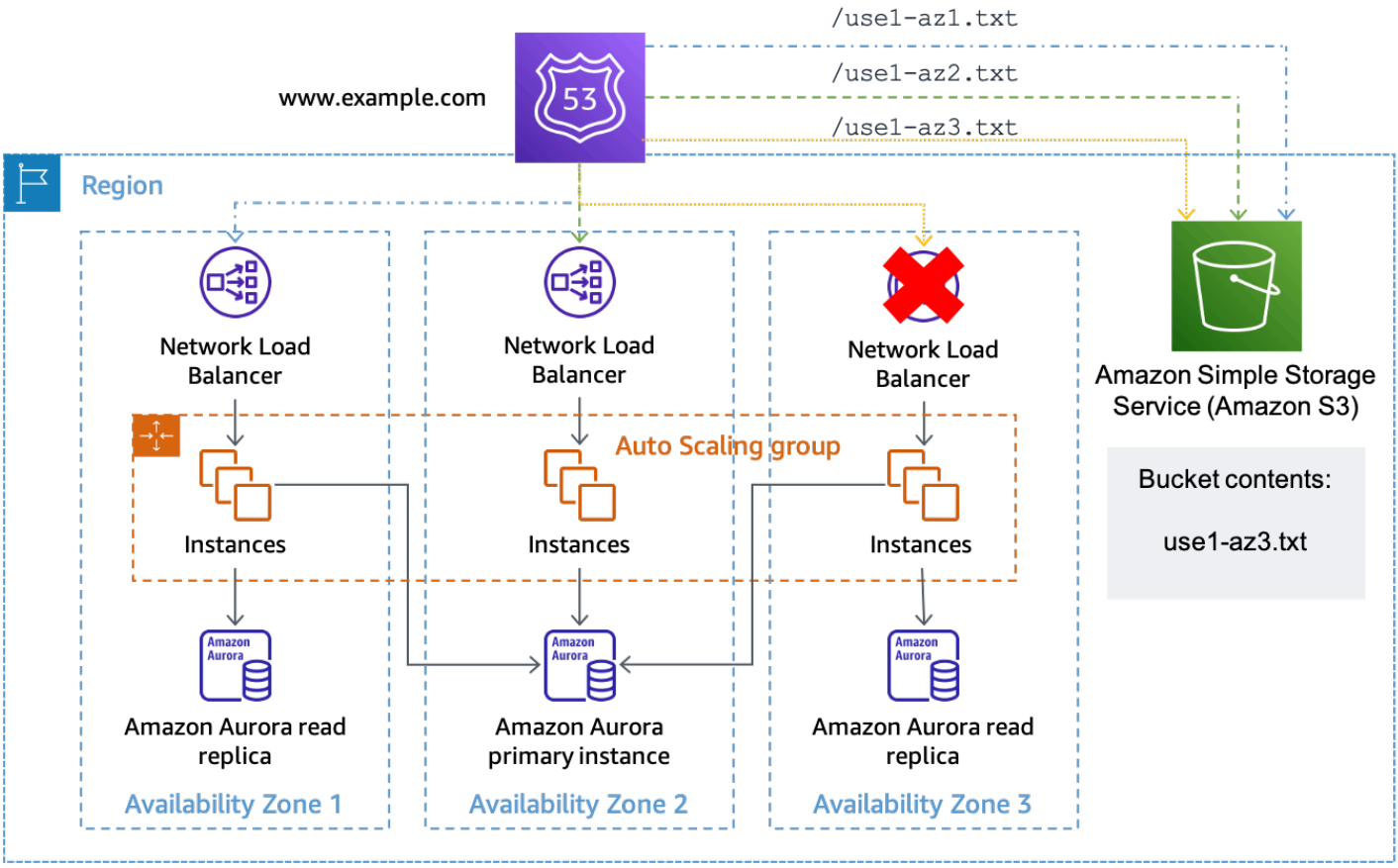
表 4：根据可用区使用 Route 53 运行状况检查时 DNS 记录的配置

运行状况检查类型：	运行状况检查类型：	运行状况检查类型：	←	运行状况检查

监控端点 协议：HTTPS ID：dddd-4444 URL：https://bname .s3.us-east-1.amazonaws.com/use1-az1.txt	监控端点 协议：HTTPS ID：eeee-5555 URL：https://bucketname .s3.us-east-1.amazonaws.com/use1-az3.txt	监控端点 协议：HTTPS ID：ffff-6666 URL：https://bname .s3.us-east-1.amazonaws.com/use1-az2.txt		
↑	↑	↑		
路由策略：加权 名称：www.example.com 类型：A（别名） 值：us-east-1b.load-balancer-name.elb.us-east-1.amazonaws.com 权重：100 评估目标运行状况：true	路由策略：加权 名称：www.example.com 类型：A（别名） 值：us-east-1a.load-balancer-name.elb.us-east-1.amazonaws.com 权重：100 评估目标运行状况：true	路由策略：加权 名称：www.example.com 类型：A（别名） 值：us-east-1c.load-balancer-name.elb.us-east-1.amazonaws.com 权重：100 评估目标运行状况：true	←	权重均匀的顶层别名 A 记录指向 NLB AZ 特定端点

假设可用区 us-east-1a 映射到我们有工作负载且要执行可用区撤离的账户 use1-az3。为 us-east-1a.load-balancer-name.elb.us-east-1.amazonaws.com 创建的资源记录集会关联

运行状况检查，来测试 URL `https://bucket-name.s3.us-east-1.amazonaws.com/use1-az3.txt`。要为 `use1-az3` 启动可用区撤离时，请使用 CLI 或 API 将名为 `use1-az3.txt` 的文件上传到存储桶。该文件不需要包含任何内容，但必须是公开的，以便 Route 53 运行状况检查进行访问。下图演示了如何使用该实现撤离 `use1-az3`。



将 Amazon S3 作为 Route 53 运行状况检查的目标

使用 API Gateway 和 DynamoDB

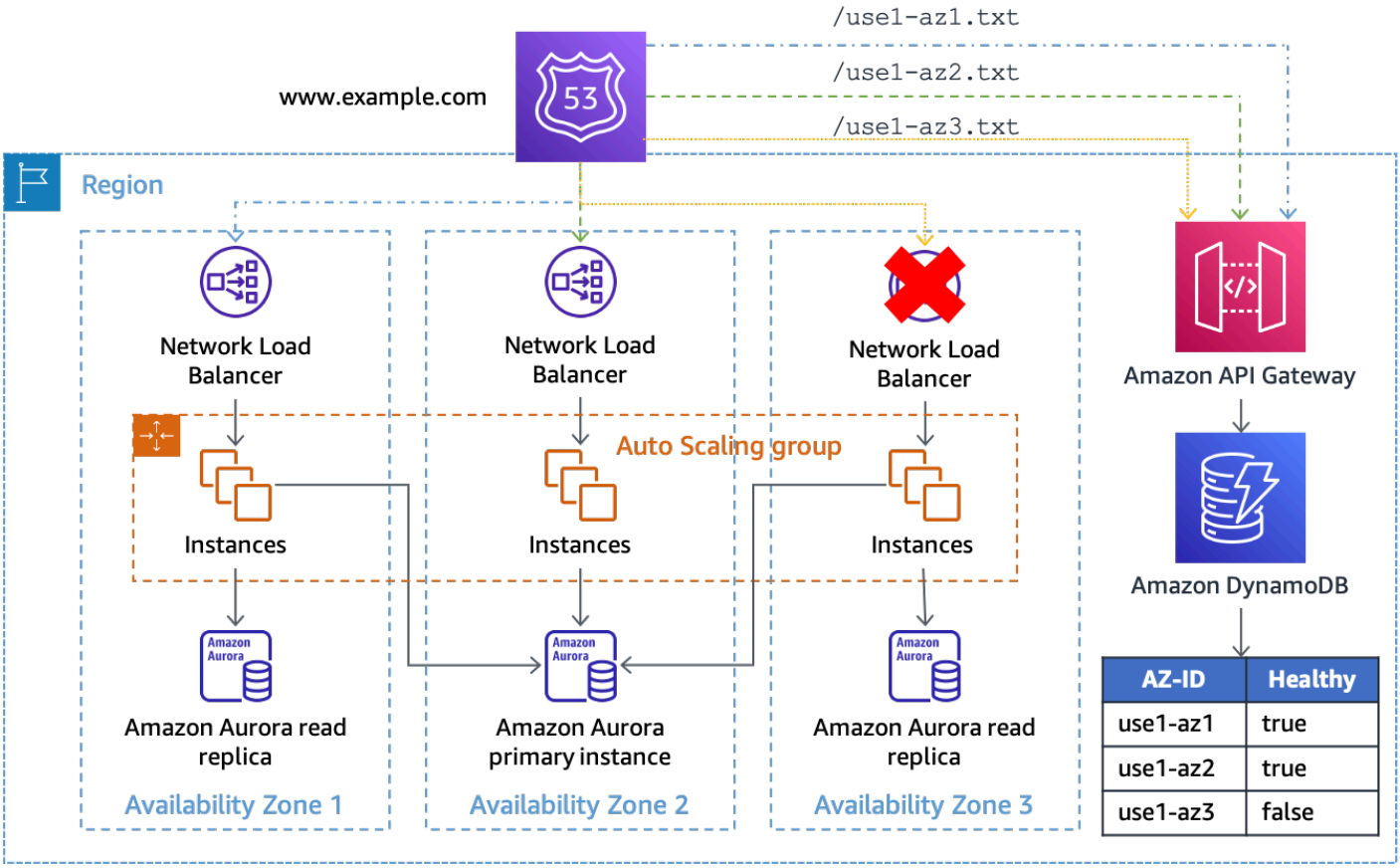
此模式的第二种实现使用 Amazon API Gateway REST API。该 API 经过配置，与 Amazon DynamoDB（其中存储了每个正在使用的可用区的状态）进行服务集成。这种实现方式比 Amazon S3 方法更灵活，但需要构建、运行和监控更多的基础设施。它也可以搭配使用 Route 53 运行状况检查以及单个主机执行的运行状况检查。

如果您将此解决方案与 NLB 或 ALB 架构搭配使用，请按照上文 Amazon S3 示例中的相同方式设置 DNS 记录，不过要更改运行状况检查路径以使用 API Gateway 端点并在 URL 路径中提供 `AZ-ID`。例如，如果将 API Gateway 自定义域配置为 `az-status.example.com`，则 `use1-az1` 的完整请求为 `https://az-status.example.com/status/use1-az1`。如果您想要启动可用区撤离，可以使用 CLI 或 API 创建或更新 DynamoDB 项目。该项目使用 `AZ-ID` 作为其主键，然后会有一个名为

Healthy 的布尔属性，用于指示 API Gateway 的响应方式。以下是 API Gateway 配置中用于做出此决定的示例代码。

```
#set($inputRoot = $input.path('$'))
#if ($inputRoot.Item.Healthy['BOOL'] == (false))
    #set($context.responseOverride.status = 500)
#end
```

如果该属性为 true（或不存在），API Gateway 会使用 HTTP 200 来响应运行状况检查；如果该属性为 false，则使用 HTTP 500 进行响应。此实现如下图所示。



将 API Gateway 和 DynamoDB 作为 Route 53 运行状况检查的目标

在此解决方案中，您需要在 DynamoDB 前面使用 API Gateway，这样您就可以公开端点，并将请求 URL 附加到 DynamoDB 的 GetItem 请求中。该解决方案非常灵活，允许您在请求中包含其他数据。例如，如果您想创建更精细的状态，例如根据应用程序，则可以将运行状况检查 URL 配置为在路径中提供应用程序 ID 或与 DynamoDB 项目匹配的查询字符串。

可用区状态端点可以集中部署，这样不同 AWS 账户 的多个运行状况检查资源都可以使用相同且一致的可用区运行状况视图（确保您的 API Gateway REST API 和 DynamoDB 表经过扩展来处理负载），且无需共享 Route 53 运行状况检查。

您也可以使用 Amazon DynamoDB 全局表和每个区域的 API Gateway REST API 副本来跨多个 AWS 区域 扩展该解决方案。这样可以避免此解决方案依赖单个区域，并提高其可用性。您可以将解决方案部署到三或五个区域并查询每个区域的可用区运行状况，这样可以使用大多数端点的结果可确保结果更客观。这样最终可以在全局表中一致地复制更新，并缓解可能阻碍端点响应的损伤。例如，如果您使用五个区域，其中三个端点报告可用区运行状况不佳，一个端点报告可用区运行状况良好，一个端点没有响应，那么您可以选择将该可用区视为运行状况不佳。您也可以创建 Route 53 计算的运行状况检查，使用 m of n 计算方法执行此逻辑，从而确定可用区的运行状况。

如果您为单个主机构建解决方案，并利用该解决方案确定可用区运行状况，那么作为替代方案，您可以使用推送通知，而不用为运行状况检查提供拉取机制。实现此操作的一种方法是通过您的使用者订阅的 SNS 话题。当您想要触发断路器时，请向 SNS 主题发布一条消息，指明哪个可用区受损。这种方法与前一种方法各有利弊。它无需创建和运行 API Gateway 基础设施，也无需执行容量管理。它还可能更快地汇总可用区状态。但是，它无法执行临时查询功能，且依赖 SNS 传输重试策略来确保每个端点都能收到通知。它还要求每个工作负载或服务都构建一种接收 SNS 通知并对其采取操作的方法。

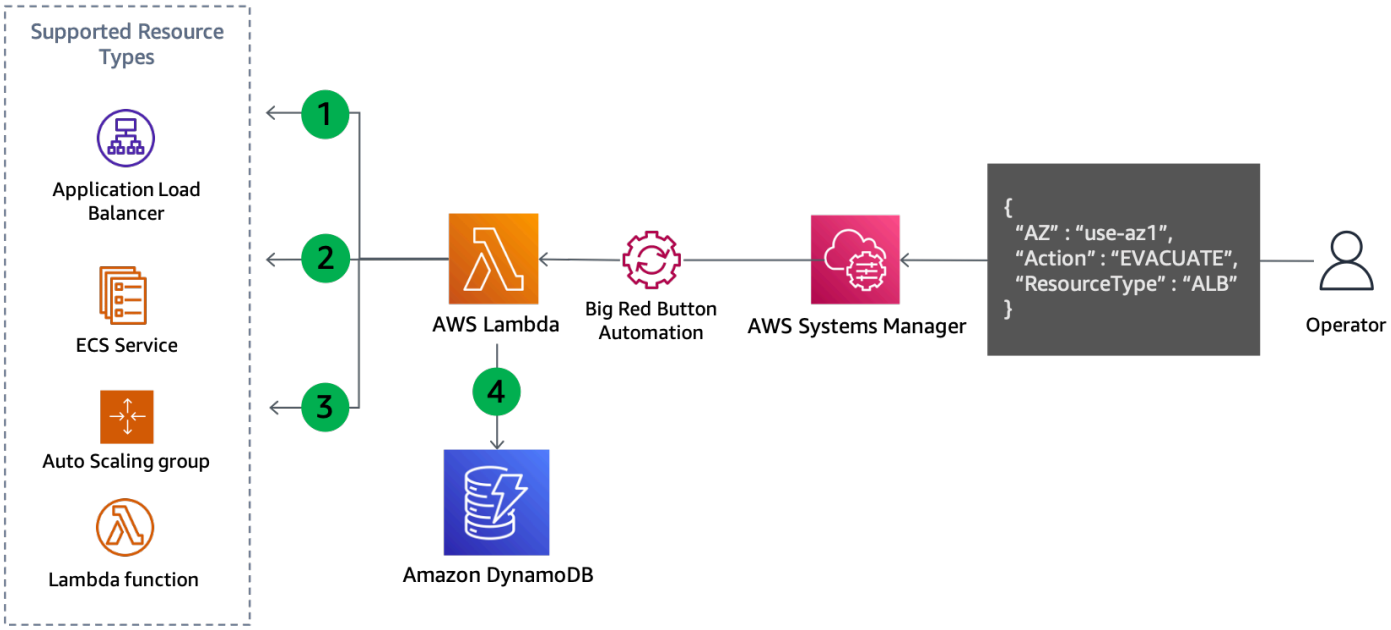
例如，启动的每个新 EC2 实例或容器都需要在引导期间使用 HTTP 端点订阅主题。然后，每个实例都需要实现软件，以监听发送通知的端点。此外，如果实例受到该事件的影响，则可能不会收到推送通知并继续工作。然而，借助拉取通知，实例将知道其拉取请求是否失败，并可以选择采取什么应对措施。

发送推送通知的第二种方法是使用长期 WebSocket 连接。Amazon API Gateway 可用于提供 WebSocket API，以供使用者连接和接收后端服务发送的消息。借助 WebSocket，实例既可以定期执行拉取操作以确保其连接正常，又可以接收低延迟推送通知。

通过控制面板控制的撤离

第一种模式使用数据面板操作来阻止在受影响的可用区中执行工作，从而缓解事件的影响。但是，您使用的架构可能不使用负载均衡器，或者无法为每台主机配置运行状况检查。或者，您可能希望通过自动扩缩或正常工作计划阻止将新容量部署到受影响的可用区。

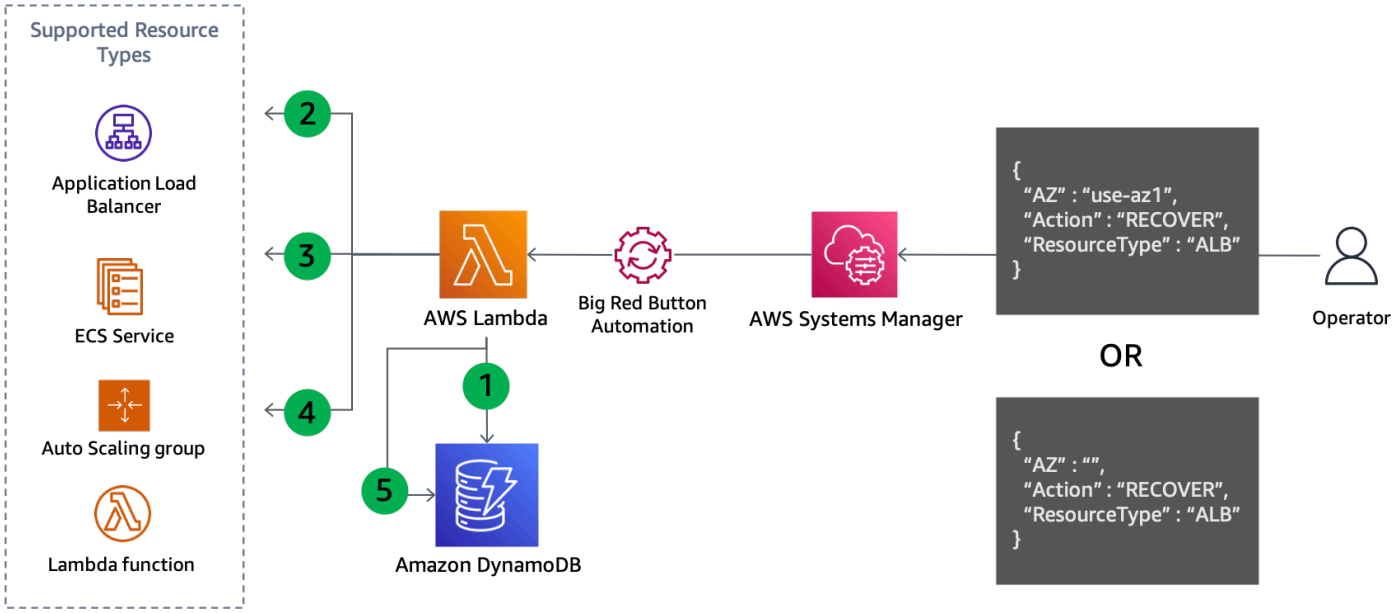
为了应对这两种情况，您需要执行控制面板操作来更新资源配置。该模式适合任意网络配置可以更新的服务，例如 EC2 Auto Scaling、Amazon ECS、Lambda 等。它需要针对每项服务编写代码，但业务逻辑遵循标准模式。代码应由响应事件的操作员在本地执行，以最大限度地减少所需的依赖项。脚本逻辑的基本流程如下图所示。



为撤离可用区而进行的控制面板更新

1. 该脚本列出了指定类型的所有资源（例如自动扩缩组、ECS 服务或 Lambda 函数）并从资源信息中检索其子网。支持的资源取决于脚本的配置。
2. 它通过将每个子网的可用区名称与映射的可用区 ID（作为输入参数提供）进行比较来确定应删除哪些子网。
3. 资源的网络配置已更新以移除已识别的子网。
4. 更新的详细信息记录在 DynamoDB 表中。可用区 ID 作为分区键进行存储，而资源 ARN 或名称则作为排序键进行存储。已删除的子网则作为字符串数组进行存储。最后，还会存储资源类型，并用作全局二级索引 (GSI) 的哈希键。

步骤 4 记录了所做的更新，因此当您准备好恢复时，这种方法还能够轻松恢复，如下图所示。



为恢复可用区撤离操作而进行的控制面板更新

恢复步骤：

1. 查询 GSI 以获取针对指定可用区（如果未指定，则为所有可用区）中指定类型的每种资源删除的子网。
2. 描述在 DynamoDB 查询中找到的每个资源以获取其当前网络配置。
3. 将当前网络配置中的子网与从 DynamoDB 查询中检索到的子网合并。
4. 使用新的子网集更新资源的网络配置。
5. 更新成功完成后，从 DynamoDB 表中删除该记录。

这种通用模式既可以阻止将工作路由到受影响的可用区，又可以阻止将新的容量部署到这些可用区。以下示例介绍如何针对不同服务实现该模式。

- Lambda — 更新函数的 VPC 配置以删除指定可用区中的子网。
- 自动扩缩组 — 从 ASG 配置中删除子网，这将替换剩余可用区中的该容量。
- Amazon ECS — 更新 ECS 服务 VPC 配置以删除子网。
- Amazon EKS — 对受影响的可用区中的节点应用污点，以驱逐现有容器组 (pod) 并防止在此安排其他容器组 (pod)。

各项服务对配置更新的反应都会有所不同。例如，Amazon ECS 将在更新后实施服务的部署配置，并触发新任务的滚动部署或蓝/绿部署。

对于某些工作负载来说，这些更新可能会过快地将工作转移到正常的可用区。尽管经过配置可在面临该故障时保持静态稳定（在剩余的可用区中预先配置足够的容量来处理受影响的可用区的工作），您可能仍希望逐步淘汰受影响可用区中的容量。

如果您计划更新已禁用跨区域负载均衡的负载均衡器的目标组（自动扩缩组）网络配置，请遵循此指南。

Auto Scaling 会使用其可用区重新平衡逻辑对这一变化做出反应。它将在其他可用区启动实例以满足您所需的容量，并终止您移除的可用区中的实例。但是，在实例终止的过程中，负载均衡器将继续在每个可用区（包括您从 ASG 中移除的可用区）之间平均分配流量。这可能会导致该可用区的剩余容量减少，直到所有实例都成功终止。这与可用区独立性中描述问题相同，涉及禁用跨区域负载均衡导致的可用区不平衡。为避免出现此类情况，您可以：

- 始终先撤离可用区，这样流量就只会在剩余的可用区之间进行分配
- 指定运行状况良好的目标最低计数进行 DNS 故障转移，以匹配该可用区所需的最低目标数量。

这将有助于确保在实例开始终止后不会将流量发送到您移除的可用区。

摘要

下表总结了所介绍的撤离模式的优缺点。

表 5：撤离模式的优缺点

方法	优点	缺点
通过数据面板控制的撤离	仅依赖数据面板操作 快速阻止在受影响的可用区内执行工作 方法灵活，可集中查看可用区运行状况	不会阻止在受影响的可用区部署容量 并非所有工作负载类型都能轻松使用这种方法
通过控制面板控制的撤离	防止在受影响的可用区部署新容量	依赖各项服务的控制面板

方法	优点	缺点
	从受影响的可用区移除现有容量	需要针对各项服务编写代码 必须针对各项服务逐个完成 需要注意不要在更新期间超出容量

在可用区撤离计划中，您可能会同时使用这两种方法。首先使用数据面板控制的撤离操作，这些操作更有可能快速成功地停止受影响可用区中的工作。然后，一旦最初的影响得到缓解，则可以执行控制面板控制的撤离操作（如果您认为有必要）。

结论

本文概述了灰色故障及其表现方式，并概述了为什么您需要建立可观测性和撤离工具，以便发生这些类型的事件时缓解其影响。在下一部分中，您回顾了多可用区可观测性以及可以检测单个可用区影响的三种方法。在最后一部分，本文介绍了两种执行可用区撤离的常规方法。第一种方法使用数据面板操作来防止将工作路由到受影响的可用区，而第二种方法则使用控制面板操作来防止在受影响的可用区中配置容量。这两种方法共同实现了可用区撤离预期达到的两个结果。

本文中描述的恢复模式很可能是更大型的监控和故障恢复解决方案的一部分。这种处理单可用区灰色故障的方法需要开展工程工作来构建检测这些故障所需的仪器以及应对这些故障的工具。但是，对于许多工作负载，与构建多区域架构相比，这种替代方法是更简单、成本更低。此外，与多区域灾难恢复相比，它可以帮助实现更低的 RPO 和 RTO（从而提高工作负载的可用性）。

附录 A – 获取可用区 ID

如果您使用 AWS .NET 开发工具包（以及其他开发工具包，例如 JavaScript）或在 EC2 实例（包括 Amazon ECS 和 Amazon EKS）上运行系统，则可以直接获取可用区 ID。

- AWS .NET 开发工具包

```
Amazon.Util.EC2InstanceMetadata.GetData(“/placement/availability-zone-id”)
```

- EC2 实例元数据服务

```
curl http://169.254.169.254/latest/meta-data/placement/availability-zone-id
```

在其他平台上，例如 Lambda 和 Fargate，您需要检索可用区名称，然后找到与可用区 ID 的映射。拥有可用区名称后，您可以通过如下方法找到可用区 ID：

```
aws ec2 describe-availability-zones --zone-names $AZ --output json
    --query ‘AvailabilityZones[0].ZoneId’
```

以下用于查找要在上面示例中使用的可用区名称的示例是使用 AWS CLI 和包 jq 在 bash 中编写的。它们需要转换为用于您的工作负载的编程语言。

- Amazon ECS - 如果实例元数据服务 (IMDS) 被主机阻止，则可以改用容器元数据文件。

```
AZ=$(cat $ECS_CONTAINER_METADATA_FILE | jq --raw-output
    .AvailabilityZone)
```

- Fargate（平台版本 1.4 或更高）

```
AZ=$(curl $ECS_CONTAINER_METADATA_URI_V4/task | jq --raw-output
    .AvailabilityZone)
```

- Lambda – 可用区不直接向函数开放。要找到它，您需要完成几步操作。为此，您需要构建一个私有 API Gateway REST 端点，用于返回请求者的 IP 地址。这将识别分配给该函数正在使用的弹性网络接口的私有 IP。
  - 调用 Lambda `GetFunction` API 来查找函数的 VPC ID。

- 调用 API Gateway 服务来获取函数的 IP。
- 借助 IP 和 VPC ID，找到关联的网络接口并提取可用区。

```
VPC_ID=$(aws lambda get-function --function-name $ AWS_LAMBDA_FUNCTION_NAME --
region $AWS_REGION --output json --query ‘Configuration.VpcConfig.VpcId’)

MY_IP=$(curl http://whats-my-private-ip.internal)

AZ=$(aws ec2 describe-network-interfaces --filters Name=private-ip-address,Values=
$MY_IP Name=vpc-id,Values=$VPC_ID --region $AWS_REGION --output json –query
 ‘NetworkInterfaces[0].AvailabilityZone’)
```

附录 B – 卡方计算示例

以下是收集错误指标并对数据执行卡方检验的示例。该代码尚未实现生产就绪，也不会执行必要的错误处理，但提供了逻辑工作原理的概念验证。您应该更新此示例以满足您的需求。

首先，Amazon EventBridge 计划的事件每分钟调用一次 Lambda 函数。该事件的内容配置有以下数据：

```
{
  "timestamp": "2023-03-15T15:26:37.527Z",
  "namespace": "multi-az/frontend",
  "metricName": "5xx",
  "dimensions": [
    { "Name": "Region", "Value": "us-east-1" },
    { "Name": "Controller", "Value": "Home" },
    { "Name": "Action", "Value": "Index" }
  ],
  "period": 60,
  "stat": "Sum",
  "unit": "Count",
  "chiSquareMetricName": "multi-az/chi-squared",
  "azs": [ "use1-az2", "use1-az4", "use1-az6" ]
}
```

这些数据用于指定检索相应的 CloudWatch 指标（如命名空间、指标名称和维度）所需的常见数据，然后发布每个可用区的卡方检验结果。使用 Python 3.9，Lambda 函数中的代码如下所示。简而言之，它收集前一分钟指定的 CloudWatch 指标，对该数据运行卡方检验，然后发布有关每个指定可用区的检验结果的 CloudWatch 指标。

```
import os
import boto3
import datetime
import copy
import json
from datetime import timedelta
from scipy.stats import chisquare
from aws_embedded_metrics import metric_scope

cw_client = boto3.client("cloudwatch", os.environ.get("AWS_REGION", "us-east-1"))
```

```
@metric_scope
def handler(event, context, metrics):
    metrics.set_property("Event", json.loads(json.dumps(event, default = str)))
    time = datetime.datetime.strptime(event["timestamp"], "%Y-%m-%dT%H:%M:%S.%fZ")

    # Round down to the previous minute
    end: datetime = roundTime(time)

    # Subtract a minute for the start
    start: datetime = end - timedelta(minutes = 1)

    # Get all the metrics that match the query
    results = get_all_metrics(event, start, end, metrics)
    metrics.set_property("MetricCounts", results)

    # Calculate the chi squared result
    chi_sq_result = chisquare(list(results.values()))
    expected = sum(list(results.values())) / len(results.values())
    metrics.set_property("ChiSquaredResult", chi_sq_result)

    # Put the chi square metrics into CloudWatch
    put_all_metrics(event, results, chi_sq_result[1], expected, start, metrics)

def get_all_metrics(detail: dict, start: datetime, end: datetime, metrics):
    """
    Gets all of the error metrics for each AZ specified
    """
    metric_query = {
        "MetricDataQueries": [
        ],
        "StartTime": start,
        "EndTime": end
    }

    for az in detail["azs"]:

        dim = copy.deepcopy(detail["dimensions"])
        dim.append({"Name": "AZ-ID", "Value": az})

        query = {
            "Id": az.replace("-", "_"),
            "MetricStat": {
                "Metric": {
                    "Namespace": detail["namespace"],
```

```
                    "MetricName": detail["metricName"],
                    "Dimensions": dim
                },
                "Period": int(detail["period"]),
                "Stat": detail["stat"],
                "Unit": detail["unit"]
            },
            "Label": az,
            "ReturnData": True
        }

        metric_query["MetricDataQueries"].append(query)

    metrics.set_property("GetMetricRequest", json.loads(json.dumps(metric_query,
default=str)))
    next_token: str = None
    results = {}

    while True:
        if next_token is not None:
            metric_query["NextToken"] = next_token

        data = cw_client.get_metric_data(**metric_query)

        if next_token is not None:
            metrics.set_property("GetMetricResult::" + next_token,
json.loads(json.dumps(data, default = str)))
        else:
            metrics.set_property("GetMetricResult", json.loads(json.dumps(data, default
= str)))

        for item in data["MetricDataResults"]:
            key = item["Id"].replace("_", "-")
            if key not in results:
              results[key] = 0

            results[key] += sum(item["Values"])

        if "NextToken" in data:
            next_token = data["NextToken"]

        if next_token is None:
            break
```

```
    return results

def put_all_metrics(detail: dict, results: dict, chi_sq_value: float, expected: float,
 timestamp: datetime, metrics):
    """
    Adds the chi squared metric for all AZs to CloudWatch
    """
    farthest_from_expected = None
    if len(results) > 0:
        keys = list(results.keys())
        farthest_from_expected = keys[0]

        for key in keys:
            if abs(results[key] - expected) > abs(results[farthest_from_expected] -
 expected):
                farthest_from_expected = key

    metric_query = {
        "Namespace": detail["namespace"],
        "MetricData": []
    }

    for az in detail["azs"]:
        dim = copy.deepcopy(detail["dimensions"])
        dim.append({"Name": "AZ-ID", "Value": az})

        query = {
            "MetricName": detail["chiSquareMetricName"],
            "Dimensions": dim,
            "Timestamp": timestamp,
        }

        if chi_sq_value <= 0.05 and az == farthest_from_expected:
            query["Value"] = 1
        else:
            query["Value"] = 0

        metric_query["MetricData"].append(query)

    metrics.set_property("PutMetricRequest", json.loads(json.dumps(metric_query,
 default = str)))

    cw_client.put_metric_data(**metric_query)
```

```
def roundTime(dt=None, roundTo=60):
   """Round a datetime object to any time lapse in seconds
   dt : datetime.datetime object, default now.
   roundTo : Closest number of seconds to round to, default 1 minute.
   """
   if dt == None : dt = datetime.datetime.now()
   seconds = (dt.replace(tzinfo=None) - dt.min).seconds
   rounding = (seconds+roundTo/2) // roundTo * roundTo
   return dt + datetime.timedelta(0,rounding-seconds,-dt.microsecond)
```

然后，您可以为每个可用区创建警报。以下示例针对 use1-az2，介绍三个连续一分钟数据点的警报，这些数据点的最大值等于 1（1 是卡方检验确定错误率存在统计学显著偏差时发布的指标）。

```
{
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
        "AlarmName": "use1-az2-chi-squared",
        "ActionsEnabled": true,
        "OKActions": [],
        "AlarmActions": [],
        "InsufficientDataActions": [],
        "MetricName": "multi-az/chi-squared",
        "Namespace": "multi-az/frontend",
        "Statistic": "Maximum",
        "Dimensions": [
            {
                "Name": "AZ-ID",
                "Value": "use1-az2"
            },
            {
                "Name": "Action",
                "Value": "Index"
            },
            {
                "Name": "Region",
                "Value": "us-east-1"
            },
            {
                "Name": "Controller",
                "Value": "Home"
            }
```

```
        ],
        "Period": 60,
        "EvaluationPeriods": 3,
        "DatapointsToAlarm": 3,
        "Threshold": 1,
        "ComparisonOperator": "GreaterThanOrEqualToThreshold",
        "TreatMissingData": "missing"
    }
}
```

您还可以创建 M（最大为 N）警报，并将这两个警报与复合警报合并在一起。您还需要为每个可用区中的每个控制器/操作组合或微服务创建相同的警报。最后，您可以将卡方复合警报添加到每个控制器/操作组合的可用区特定警报中，如 使用异常值检测进行故障检测 所示。

贡献者

本文档的贡献者包括：

- Michael Haken，Amazon Web Services 首席解决方案架构师

文档修订

如需获取有关本白皮书更新的通知，请订阅 RSS 源。

变更	说明	日期
已更新白皮书	新增了额外的可观测性指导，并使用了新的可用区转移功能。	2023 年 7 月 11 日
初次发布	白皮书首次发布。	2022 年 3 月 2 日

> **Note**
>
> 要订阅 RSS 更新，您必须为当前使用的浏览器启用 RSS 插件。

注意事项

客户有责任对本文档中的信息进行单独评测。本文档：(a) 仅供参考，(b) 代表当前的 AWS 产品和实践，如有更改，恕不另行通知，以及 (c) 不构成 AWS 及其附属公司、供应商或许可方的任何承诺或保证。AWS 产品或服务“按原样”提供，不附带任何明示或暗示的保证、陈述或条件。AWS 对其客户承担的责任和义务受 AWS 协议制约，本文档不是 AWS 与客户直接协议的一部分，也不构成对该协议的修改。

AWS 术语表

有关最新的 AWS 术语，请参阅《AWS 词汇表参考》中的 AWS 词汇表。

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。