Detecting and Mitigating Gray Failures

Advanced Multi-AZ Resilience Patterns

Advanced Multi-AZ Resilience Patterns: Detecting and Mitigating Gray Failures

Table of Contents

Advanced Multi-AZ Resilience Patterns

Publication date: **July 11, 2023** (*Document revisions*)

Many customers run their workloads in highly available, multi-Availability Zone (AZ) configurations. These architectures perform well during binary failure events, but often encounter problems with *gray* failures. The manifestations of this type of failure can be subtle, and defy quick and definitive detection. This paper provides guidance on how to instrument workloads to detect impact from gray failures that are isolated to a single Availability Zone, and then take action to mitigate that impact in the Availability Zone.

Introduction

The purpose of this document is to help you more effectively implement resilient multi-AZ architectures. One of the best practices for building resilient systems in Amazon Virtual Private Cloud (VPC) networks is to deploy each workload to multiple Availability Zones.

An Availability Zone is one or more discrete data centers with redundant power, networking, and connectivity. Using multiple Availability Zones allows you to operate workloads that are more highly available, fault tolerant, and scalable than would be possible from a single data center.

Many AWS services, such as Amazon Elastic Compute Cloud (EC2) Auto Scaling or Amazon Relational Database Service (Amazon RDS), provide a multi-AZ configuration. These services don't require you to build any additional observability or failover tooling. They make workloads resilient to easily detectable binary failure modes within an AWS Region that affect a single Availability Zone. This could be complete physical hardware failure, power loss, or a latent software bug affecting a majority of resources.

But there is another category of failures termed *gray failures*, whose manifestations are subtle and defy quick and definitive detection. This in turn results in longer times to mitigate the impact caused by the failure. This paper focuses on the impacts gray failures can have on multi-AZ architectures, how to detect them, and, finally, how to mitigate them.

> ⓘ The guidance provided in this whitepaper is mostly applicable to specific classes of workloads that:
>
> - Primarily use zonal AWS services

- Need to improve single Region resilience
- Are willing to make a significant investment to build the required observability and resilience patterns

In these workloads, you might not be willing to make some, or all, of the tradeoffs presented in ???, or not have the option to use multiple Regions. These types of workloads are likely to represent a small subset of your overall portfolio and hence this guidance should be considered at the workload level versus at the platform level.

Gray failures

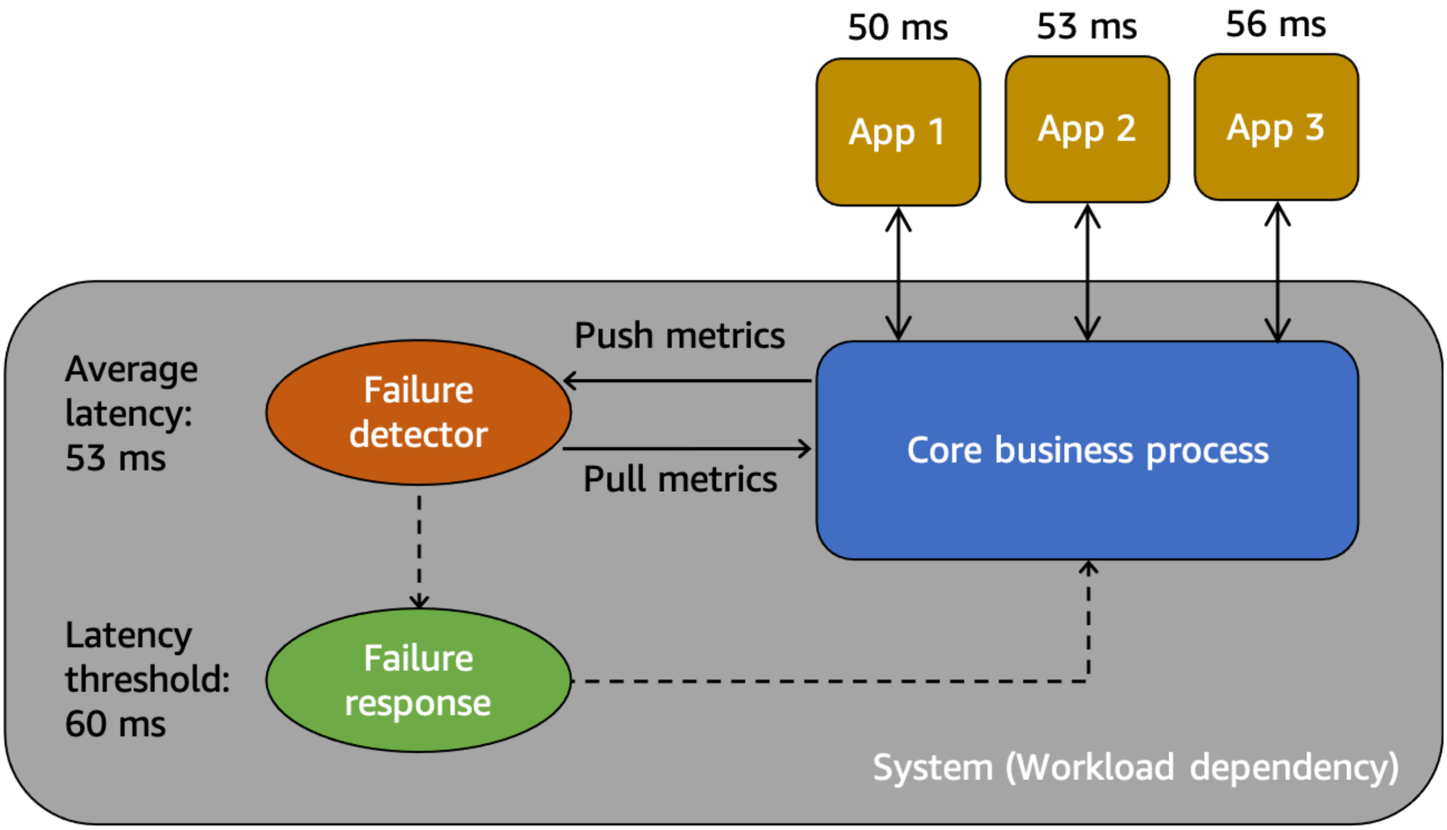
Gray failures are defined by the characteristic of *differential observability*, meaning that different entities observe the failure differently. Let’s define what this means.

Differential observability

The workloads that you operate typically have dependencies. For example, these can be the AWS cloud services that you use to build your workload or a third-party identity provider (IdP) you use for federation. Those dependencies almost always implement their own observability, recording metrics about errors, availability, and latency among other things that are generated by their customer usage. When a threshold is crossed for one of these metrics, the dependency usually takes some action to correct it.

These dependencies usually have multiple consumers of their services. Consumers also implement their own observability and record metrics and logs about their interactions with their dependencies, recording things like how much latency there is in disk reads, how many API requests failed, or how long a database query took.

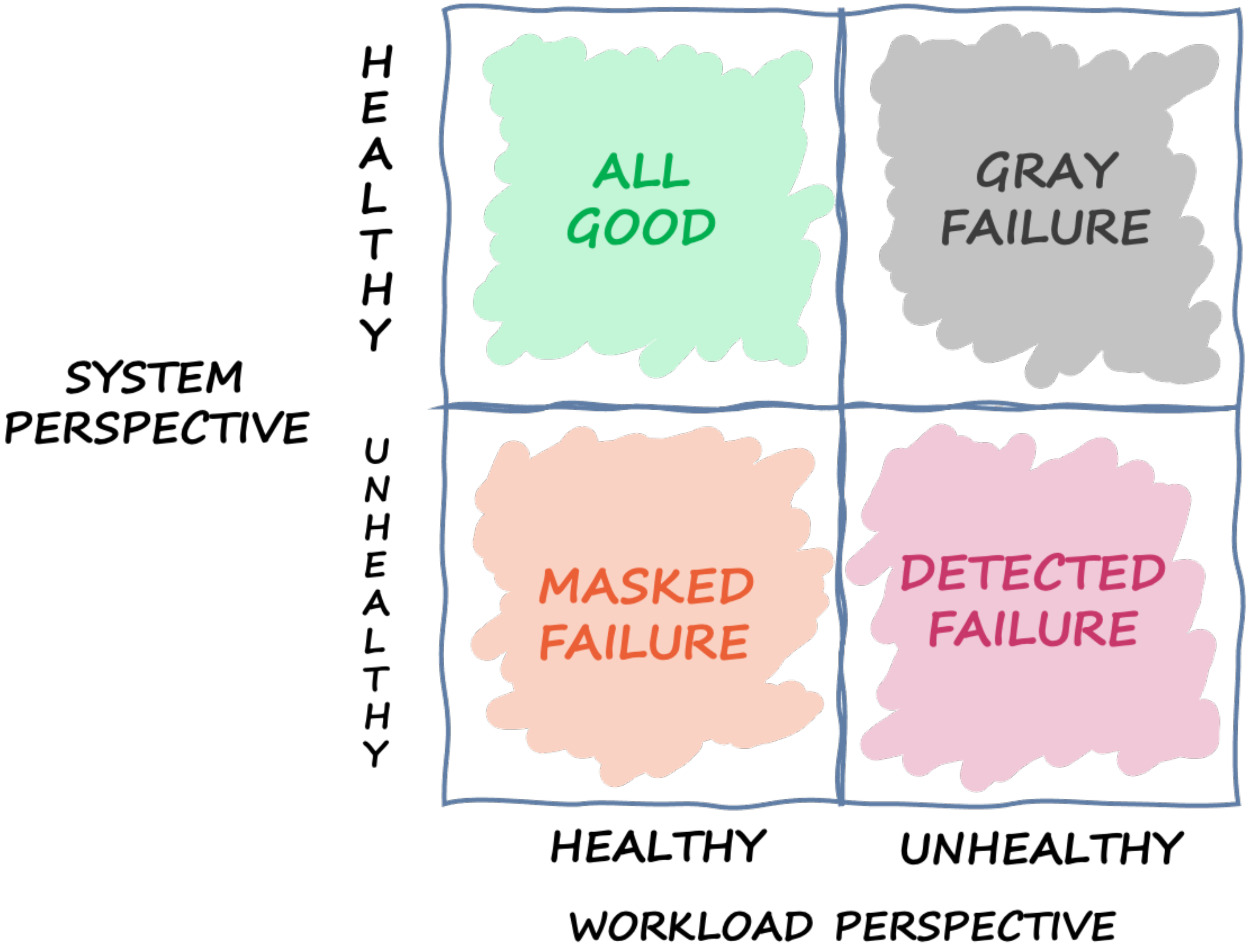
These interactions and measurements are depicted in an abstract model in the following figure.



An abstract model for understanding gray failures

First, we have the *system*, which is a dependency for consumers App 1, App 2, and App 3 in this scenario. The system has a failure detector that examines metrics created from the core business process. It also has a failure response mechanism to mitigate or correct problems that are observed by the failure detector. The system sees an overall average latency of 53 ms and has set a threshold to invoke the failure response mechanism when average latency exceeds 60 ms. App 1, App 2, and App 3 are also making their own observations about their interaction with the system, recording an average latency of 50 ms, 53 ms, and 56 ms respectively.

Differential observability is the situation where one of the system consumers detects that the system is unhealthy, but the system's own monitoring does not detect the problem or the impact does not cross an alarm threshold. Let's imagine that App 1 starts experiencing an average latency of 70 ms instead of 50ms. App 2 and App 3 don't see a change in their average latencies. This increases the average latency of the underlying system to 59.66 ms, but this does not cross the latency threshold to activate the failure response mechanism. However, App 1 sees a 40% increase in latency. This could impact its availability by exceeding the configured client timeout for App 1, or it may cause cascading impacts in a longer chain of interactions. From the perspective of App 1, the underlying system it depends on is unhealthy, but from the perspective of the system itself as well App 2 and App 3, the system is healthy. The following figure summarizes these different perspectives.



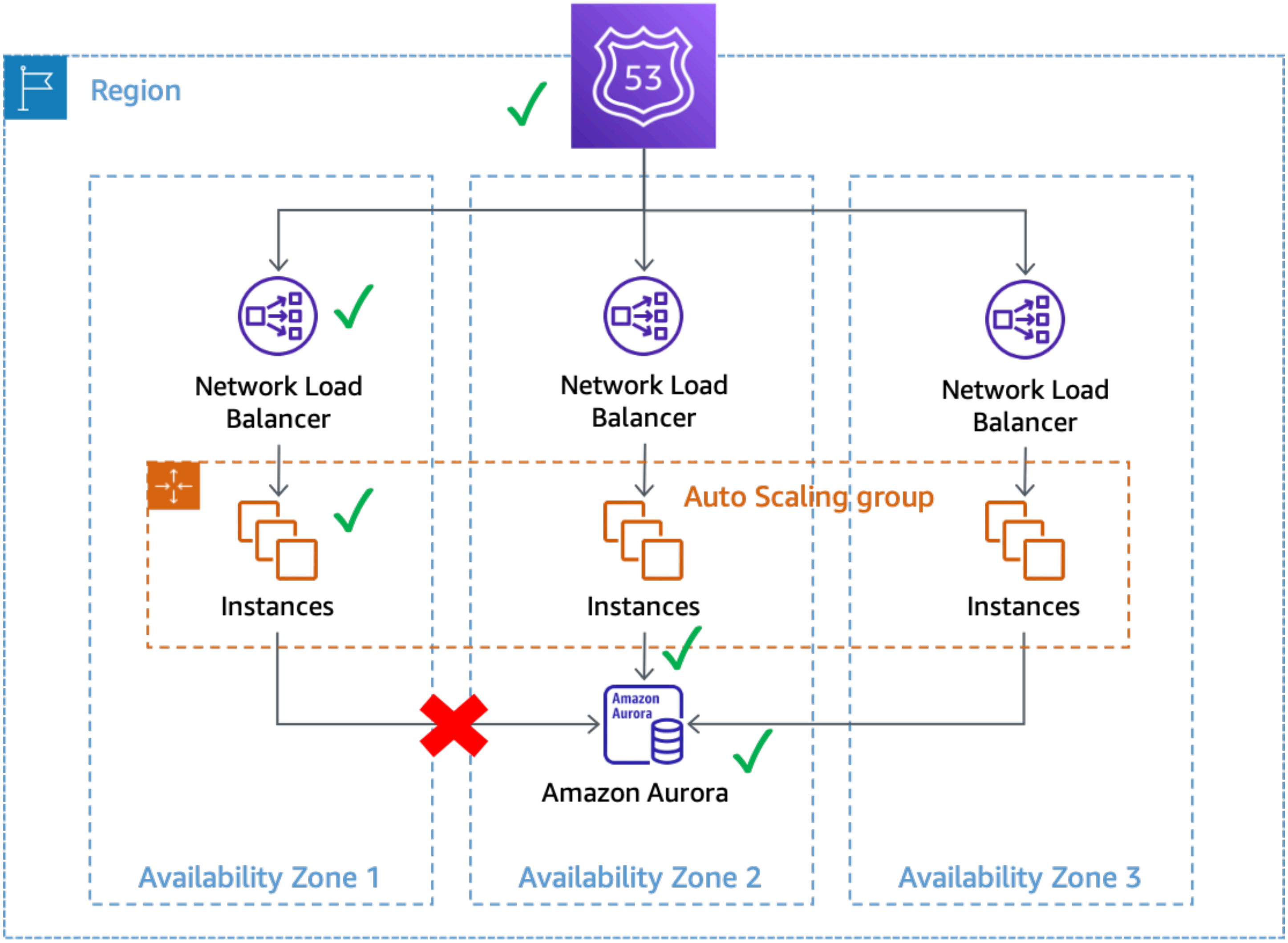
A quadrant defining the different states a system can be in based on different perspectives

The failure can also traverse this quadrant. An event could start as a gray failure, then become a detected failure, then move to a masked failure, and then maybe back to a gray failure. There's not a defined cycle, and there's almost always a chance of failure recurrence until its root cause is addressed.

The conclusion we draw from this is that workloads can't always rely on the underlying system to detect and mitigate the failure. No matter how sophisticated and resilient the underlying system is, there will always be the chance that a failure could go undetected or stay under the reaction threshold. The consumers of that system, like App 1, need to be equipped to both quickly detect and mitigate the impact a gray failure causes. This requires building observability and recovery mechanisms for these situations.

Gray failure example

Gray failures can have impact for multi-AZ systems in AWS. For example, take a fleet of Amazon EC2 instances in an Auto Scaling group deployed across three Availability Zones. They all connect to an Amazon Aurora database in one Availability Zone. Then, a gray failure occurs that impacts networking between Availability Zone 1 and Availability Zone 2. The result of this impairment is that a percentage of new and existing database connections from instances in Availability Zone 1 fail. This situation is shown in the following figure.



A gray failure that impacts database connections from instances in Availability Zone 1

In this example, Amazon EC2 sees the instances in Availability Zone 1 as healthy because they continue to pass system and instance status checks. Amazon EC2 Auto Scaling also doesn't detect direct impact to any Availability Zone, and continues to launch capacity in the configured Availability Zones. The Network Load Balancer (NLB) also sees the instances behind it as healthy as do the Route 53 health checks that are performed against the NLB endpoint. Similarly, Amazon Relational Database Service (Amazon RDS) sees the database cluster as healthy and does not trigger an automated failover. We have many different services that all see their service and resources as healthy, but the workload detects a failure that impacts its availability. This is a gray failure.

Responding to gray failures

When you experience a gray failure in your AWS environment, you generally have three available options:

- Do nothing and wait for the impairment to end.
- If the impairment is isolated to a single Availability Zone, evacuate that Availability Zone.
- Failover to another AWS Region and use the benefits of AWS Regional isolation to mitigate the impact.

Many AWS customers are fine with option one for a majority of their workloads. They accept having a possibly extended Recovery Time Objective (RTO) with the tradeoff that they haven’t had to build additional observability or resilience solutions. Other customers choose to implement the third option, Multi-Region Disaster Recovery (DR), as their mitigation plan for a various number of failure modes. Multi-Region architectures can work well in these scenarios. However, there are a few tradeoffs when using this approach (refer to AWS Multi-Region Fundamentals for a full discussion about multi-Region considerations).

First, building and operating a multi-Region architecture can be a challenging, complex, and potentially expensive endeavor. Multi-Region architectures require careful consideration of which DR strategy you select. It might not be fiscally viable to implement a multi-Region active-active DR solution just to handle zonal impairments, while a backup and restore strategy might not meet your resilience requirements. Additionally, multi-Region failovers must be continuously practiced in production so that you are confident they will work when needed. This all requires a lot of dedicated time and resources to build, operate, and test.

Second, data replication across AWS Regions using AWS services today is all done asynchronously. Asynchronous replication can result in data loss. This means that during a Regional failover, there is a chance for some amount of data loss and inconsistency. Your tolerance to the amount of data loss is defined as your Recovery Point Objective (RPO). Customers, for whom strong data consistency is a requirement, have to build reconciliation systems to fix these consistency issues when the primary Region is available again. Or, they have to build their own synchronous replication or dual-write systems, which can have significant impacts on response latency, cost, and complexity. They also make the secondary Region a hard dependency for every transaction, which can potentially reduce the availability of the overall system.

Finally, for many workloads using an active/standby approach, there is a non-zero amount of time required to perform the failover to another Region. Your portfolio of workloads might need to be brought down in the primary Region in a specific order, need to drain connections, or stop specific processes. Then, the services might need to be brought back up in a specific order. New resources might also need be provisioned or require time to pass required health checks before being brought into service. This failover process can be experienced as a period of complete unavailability. This is what RTOs are concerned with.

Inside a Region, many AWS services offer strongly consistent data persistence. Amazon RDS multi-AZ deployments use synchronous replication. Amazon Simple Storage Service (Amazon S3) offers strong read-after-write consistency. Amazon Elastic Block Storage (Amazon EBS) offers multi-volume crash consistent snapshots. Amazon DynamoDB can perform strongly consistent reads. These features can help you achieve a lower RPO (in most cases a zero RPO) in a single Region than you can in multi-Region architectures.

Evacuating an Availability Zone can have a lower RTO than a multi-Region strategy, because your infrastructure and resources are already provisioned across Availability Zones. Instead of needing to carefully order services being brought down and back up, or draining connections, multi-AZ architectures can continue operating in a static way when an Availability Zone is impaired. Instead of a period of complete unavailability that can occur during a Regional failover, during an Availability Zone evacuation, many systems might see only a slight degradation, as work is shifted to the remaining Availability Zones. If the system has been designed to be statically stable to an Availability Zone failure (in this case, that would mean having capacity pre-provisioned in the other Availability Zones to absorb the load), customers of the workload might not see impact at all.

> ⓘ It's possible that the impairment of a single Availability Zone impacts one or more AWS Regional services in addition to your workload. If you observe Regional impact, you should

treat the event as a Regional service impairment although the source of that impact is from a single Availability Zone. Evacuating an Availability Zone will not mitigate this type of problem. Use the response plans you have in place to respond to a Regional service impairment when this occurs.

The rest of this document focuses on the second option, evacuating the Availability Zone, as a way to achieve lower RTOs and RPOs for single-AZ gray failures. These patterns can help achieve better value and efficiency of multi-AZ architectures and, for most classes of workloads, can reduce the need to create multi-Region architectures to handle these types of events.

Multi-AZ observability

To be able to evacuate an Availability Zone during an event that is isolated to a single Availability Zone, you first must be able to detect that the failure is, in fact, isolated to a single Availability Zone. This requires high-fidelity visibility into how the system is behaving in each Availability Zone. Many AWS services provide out-of-the-box metrics that provide operational insights about your resources. For example, Amazon EC2 provides numerous metrics such as CPU utilization, disk reads and writes, and network traffic in and out.

However, as you build workloads that use these services, you need more visibility than just those standard metrics. You want visibility into the customer experience being provided by your workload. Additionally, you need your metrics to be aligned to the Availability Zones where they are being produced. This is the insight you need to detect differentially observable gray failures. That level of visibility requires instrumentation.

Instrumentation requires writing explicit code. This code should do things such as record how long tasks take, count how many items succeeded or failed, collect metadata about the requests, and so on. You also need to define thresholds ahead of time to define what is considered normal and what isn't. You should outline objectives and different severity thresholds for latency, availability, and error counts in your workload. The Amazon Builders' Library article Instrumenting distributed systems for operational visibility provides a number of best practices.

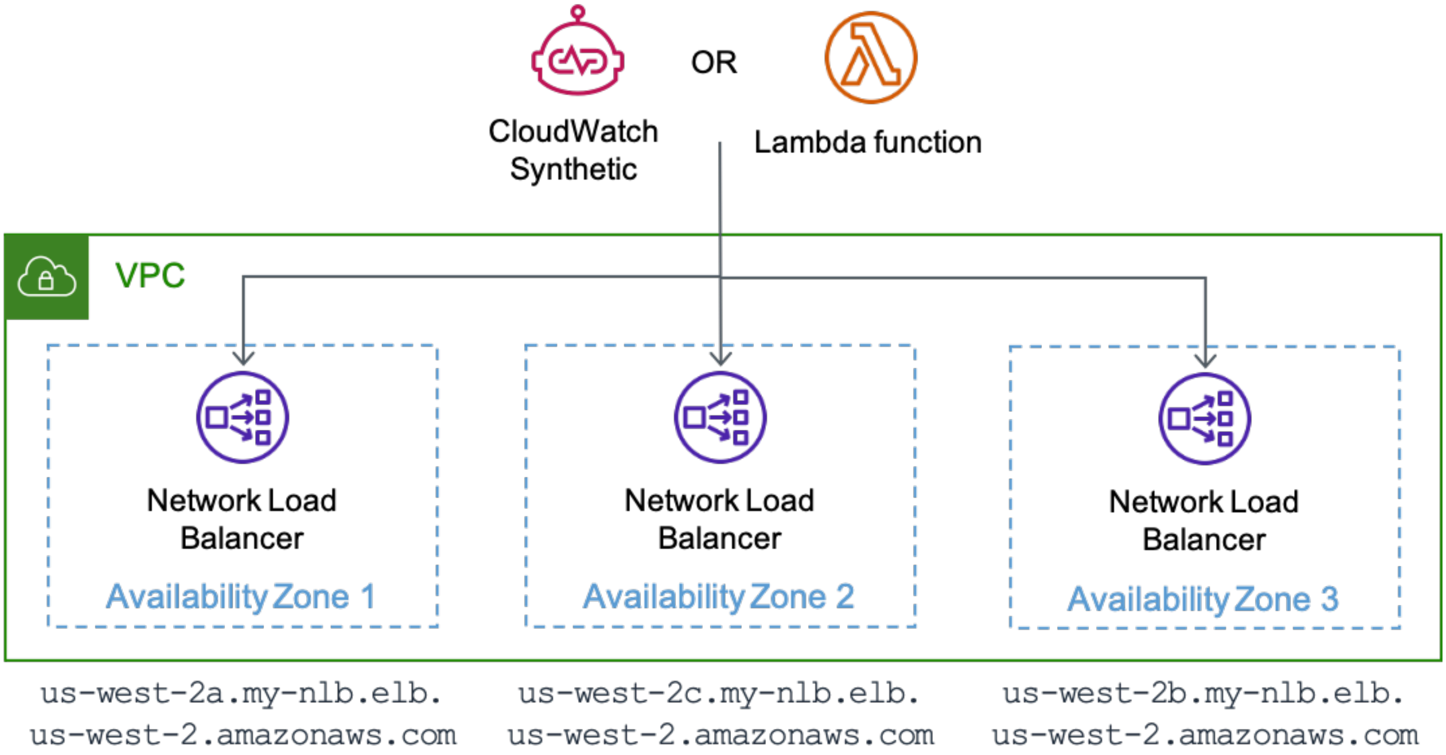
Metrics should both be generated from the server-side as well as the client-side. A best practice for generating client-side metrics and understanding the customer experience is using canaries, software that regularly probes your workload and records metrics.

In addition to producing these metrics, you also need to understand their context. One way to do this is by using dimensions. Dimensions give a metric a unique identity, and help explain what the metrics are telling you. For metrics that are used to identify failure in your workload (for example, latency, availability, or error count), you need to use dimensions that align to your fault isolation boundaries.

For example, if you are running a web service in one Region, across multiple Availability Zones, using a Model-view-controller (MVC) web framework, you should use `Region`, `Availability Zone ID`, `Controller`, `Action`, and `InstanceId` as the dimensions for your dimension sets (if you were using microservices, you might use the service name and HTTP method instead of the controller and action names). This is because you expect different types of failures to be isolated by these boundaries. You wouldn't expect a bug in your web service's code that affects its ability to list products to also impact the home page. Similarly, you wouldn't expect a full EBS volume on a single EC2 instance to affect other EC2 instances from serving your web content. The Availability Zone ID dimension is what enables you to identify Availability Zone-related impacts consistently across AWS accounts. You can find the Availability Zone ID in your workloads in a number of different ways. Refer to Appendix A – Getting the Availability Zone ID for some examples.

While this document mainly uses Amazon EC2 as the compute resource in the examples, `InstanceId` could be replaced with a container ID for Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS) compute resources as components of your dimensions.

Your canaries can also use `Controller`, `Action`, `AZ-ID`, and `Region` as dimensions in their metrics if you have zonal endpoints for your workload. In this case, align your canaries to run in the Availability Zone that they are testing. This ensures that if an isolated Availability Zone event is impacting the Availability Zone in which your canary is running, it doesn't record metrics that make a different Availability Zone it is testing appear unhealthy. For example, your canary can test each zonal endpoint for a service behind a Network Load Balancer (NLB) or Application Load Balancer (ALB) using its zonal DNS names.



A canary running on CloudWatch Synthetics or an AWS Lambda function testing each zonal endpoint of an NLB

By producing metrics with these dimensions, you can establish Amazon CloudWatch alarms that notify you when changes in availability or latency occur within those boundaries. You can also quickly analyze that data using dashboards. To use both metrics and logs efficiently, Amazon CloudWatch offers the embedded metric format (EMF) that enables you to embed custom metrics with log data. CloudWatch automatically extracts the custom metrics so you can visualize and alarm on them. AWS provides several client libraries for different programming languages that make it easy to get started with EMF. They can be used with Amazon EC2, Amazon ECS, Amazon EKS, AWS Lambda, and on-premises environments. With metrics embedded into your logs, you can also use Amazon CloudWatch Contributor Insights to create time series graphs that display contributor data. In this scenario, we could display data grouped by dimensions like `AZ-ID`, `InstanceId`, or `Controller` as well as any other field in the log like `SuccessLatency` or `HttpResponseCode`.

```
{
  "_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
}
```

This log has three sets of dimensions. They progress in order of granularity, from instance to Availability Zone to Region (`Controller` and `Action` are always included in this example). They support creating alarms across your workload that indicate when there is impact to a specific controller action in a single instance, in a single Availability Zone, or within a whole AWS Region. These dimensions are used for the count of 2xx, 3xx, 4xx, and 5xx HTTP response metrics, as well as the latency for successful request metrics (if the request failed, it would also record a metric for failed request latency). The log also records other information such as the HTTP path, the source IP of the requestor, and whether this request required the local cache to be refreshed. These data points can then be used to calculate the availability and latency of each API the workload provides.

A note on using HTTP response codes for availability metrics

Typically, you can consider 2xx and 3xx responses as successful, and 5xx as failures. 4xx response codes fall somewhere in the middle. Usually, they are produced due to a client error. Maybe a parameter is out of range leading to a 400 response, or they're requesting something that doesn't exist, resulting in a 404 response. You wouldn't count these responses against your workload's availability. However, this could also be the result of a bug in the software.

For example, if you've introduced stricter input validation that rejects a request that would have succeeded before, the 400 response might count as a drop in availability. Or maybe you're throttling the customer and returning a 429 response. While throttling a customer protects your service to maintain its availability, from the customer's perspective, the service isn't available to process their request. You'll need to decide whether or not 4xx response codes are part of your availability calculation.

While this section has outlined using CloudWatch as a way to collect and analyze metrics, it's not the only solution you can use. You might choose to also send metrics into Amazon Managed Service for Prometheus and Amazon Managed Grafana, an Amazon DynamoDB table, or use a third-party monitoring solution. The key is that the metrics your workload produces must contain context about the fault isolation boundaries of your workload.

With workloads that produce metrics with dimensions aligned to fault isolation boundaries, you can create observability that detects Availability Zone isolated failures. The following sections describe three complimentary approaches for detecting failures that arise from the impairment of a single Availability Zone.

Topics

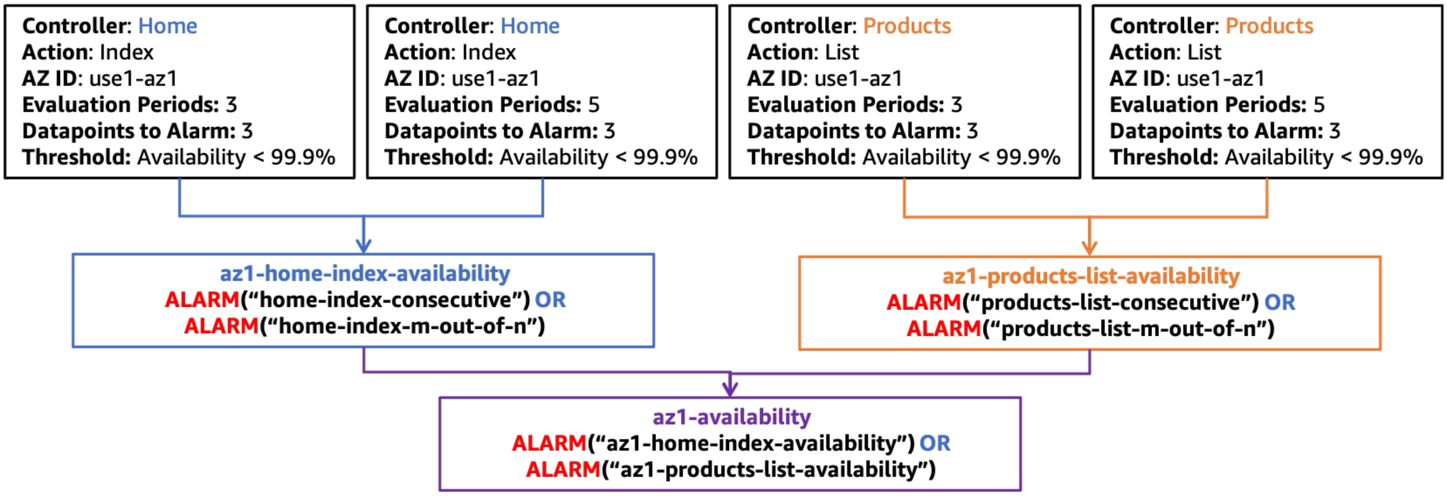
Failure detection with CloudWatch composite alarms

In CloudWatch metrics, each dimension set is a unique metric, and you can create a CloudWatch alarm on each one. You can then create Amazon CloudWatch composite alarms to aggregate these metrics.

In order to accurately detect impact, the examples in this paper will use two different CloudWatch alarm structures for each dimension set they alarm on. Each alarm will use a **Period** of one-minute, meaning the metric is evaluated once per minute. The first approach is going to use three consecutive breaching data points by setting the **Evaluation Periods** and **Datapoints to Alarm** to three, meaning impact for three minutes total. The second approach is going to use an "M out of N" when any 3 data points in a five-minute window are breaching by setting the **Evaluation Periods** to five and **Datapoints to Alarm** to three. This provides an ability to detect a constant signal, as well as one that fluctuates over a short time. The time durations and number of data points contained here are a suggestion, use values that make sense for your workloads.

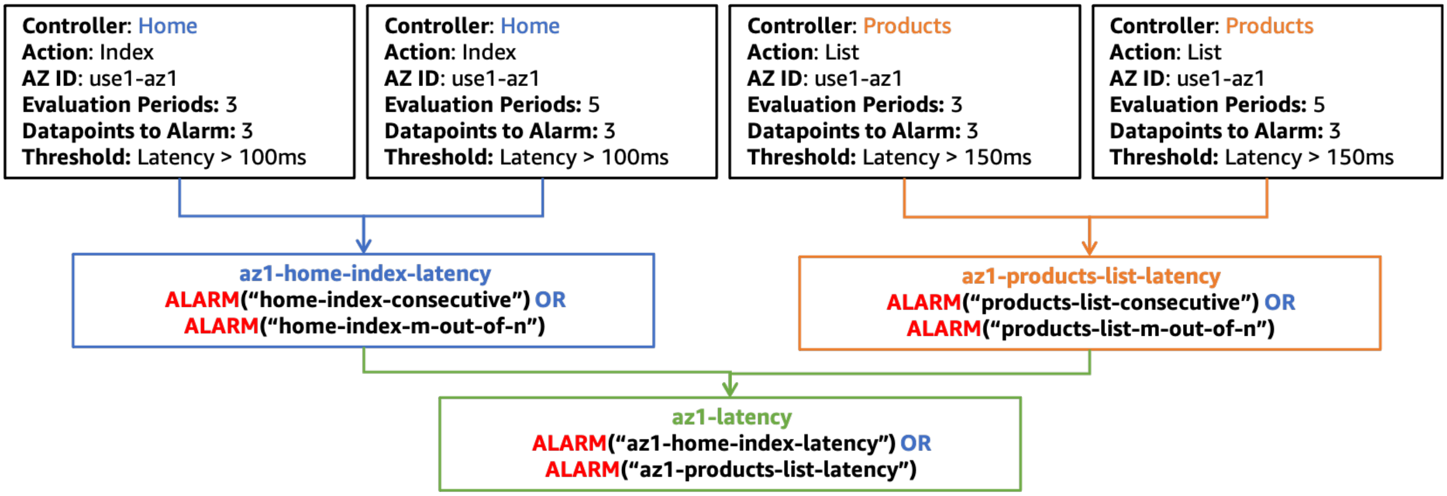
Detect impact in a single Availability Zone

Using this construct, consider a workload that uses `Controller`, `Action`, `InstanceId`, `AZ-ID`, and `Region` as dimensions. The workload has two controllers, Products and Home, and one action per controller, List and Index respectively. It operates in three Availability Zones in the `us-east-1` Region. You would create two alarms for availability for each `Controller` and `Action` combination in each Availability Zone as well as two alarms for latency for each. Then, you can optionally choose to create a composite alarm for availability for each `Controller` and `Action` combination. Finally, you create a composite alarm that aggregates all of the availability alarms for the Availability Zone. This is shown in the following figure for a single Availability Zone, `use1-az1`, using the optional composite alarm for each `Controller` and `Action` combination (similar alarms would exist for the `use1-az2` and `use1-az3` Availability Zones as well, but are not shown for simplicity).


Controller: Home
Action: Index
AZ ID: use1-az1
Evaluation Periods: 3
Datapoints to Alarm: 3
Threshold: Availability < 99.9%
Controller: Home
Action: Index
AZ ID: use1-az1
Evaluation Periods: 5
Datapoints to Alarm: 3
Threshold: Availability < 99.9%
Controller: Products
Action: List
AZ ID: use1-az1
Evaluation Periods: 3
Datapoints to Alarm: 3
Threshold: Availability < 99.9%
Controller: Products
Action: List
AZ ID: use1-az1
Evaluation Periods: 5
Datapoints to Alarm: 3
Threshold: Availability < 99.9%
az1-home-index-availability
ALARM("home-index-consecutive") OR
ALARM("home-index-m-out-of-n")
az1-products-list-availability
ALARM("products-list-consecutive") OR
ALARM("products-list-m-out-of-n")
az1-availability
ALARM("az1-home-index-availability") OR
ALARM("az1-products-list-availability")

Composite alarm structure for availability in `use1-az1`

You would also build a similar alarm structure for latency as well, shown in the next figure.

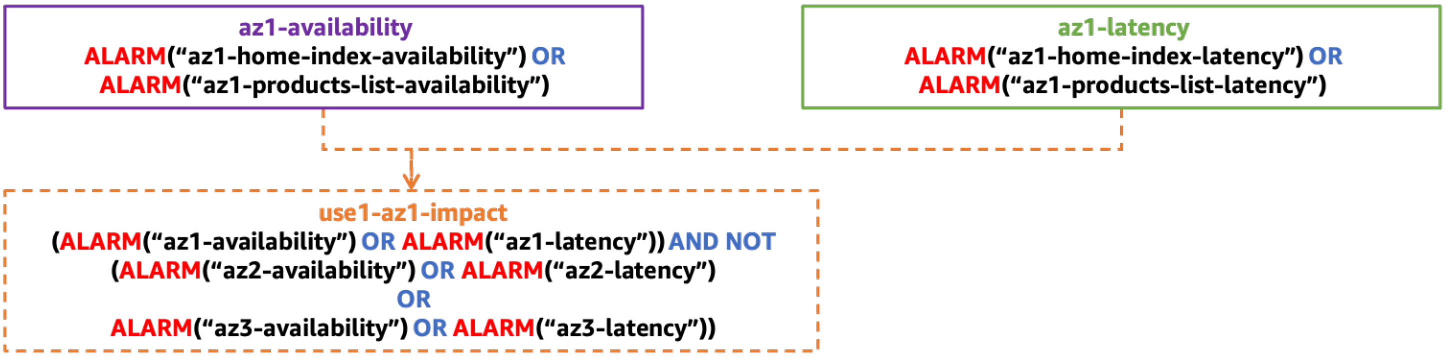


Composite alarm structure for latency in `use1-az1`

For the remainder of the figures in this section, only the `az1-availability` and `az1-latency` composite alarms will be shown at the top level. These composite alarms, `az1-availability` and `az1-latency`, will tell you if either availability drops below or latency rises above defined thresholds in a particular Availability Zone for any part of your workload. You might also want to consider measuring throughput to detect impact that prevents your workload in a single Availability Zone from receiving work. You can integrate alarms produced from the metrics emitted by your canaries into these composite alarms as well. That way, if either the server-side or client-side see impacts in availability or latency, the alarm will create an alert.

Ensure the impact isn't Regional

Another set of composite alarms can be used to ensure that only an isolated Availability Zone event causes the alarm to be activated. This is performed by ensuring that an Availability Zone composite alarm is in the `ALARM` state while the composite alarms for the other Availability Zones are in the `OK` state. This will result in one composite alarm per Availability Zone that you use. An example is shown in the following figure (remember that there are alarms for latency and availability in `use1-az2` and `use1-az3`, `az2-latency`, `az2-availability`, `az3-latency`, and `az3-availability`, that are not pictured for simplicity).


az1-availability
ALARM("az1-home-index-availability") OR
ALARM("az1-products-list-availability")
az1-latency
ALARM("az1-home-index-latency") OR
ALARM("az1-products-list-latency")
use1-az1-impact
(ALARM("az1-availability") OR ALARM("az1-latency")) AND NOT
(ALARM("az2-availability") OR ALARM("az2-latency")
OR
ALARM("az3-availability") OR ALARM("az3-latency"))

Composite alarm structure to detect impact isolated to a single AZ

Ensure the impact isn't caused by a single instance

A single instance (or a small percentage of your overall fleet) can cause disproportionate impact to availability and latency metrics that could make the whole Availability Zone appear to be affected, when in fact it is not. It is faster and just as effective to remove a single problematic instance than evacuate an Availability Zone.

Instances and containers are typically treated as ephemeral resources, frequently replaced with services such as AWS Auto Scaling. It's difficult to create a new CloudWatch alarm every time a new instance is created (but certainly possible using Amazon EventBridge or Amazon EC2 Auto Scaling lifecycle hooks). Instead, you can use CloudWatch Contributor Insights to identify the quantity of contributors to availability and latency metrics.

As an example, for an HTTP web application, you can create a rule to identify top contributors for 5xx HTTP responses in each Availability Zone. This will identify which instances are contributing to a drop in availability (our availability metric defined above is driven by the presence of 5xx errors). Using the EMF log example, create a rule using a key of `InstanceId`. Then, filter the log by the `HttpResponseCode` field. This example is a rule for the `use1-az1` Availability Zone.

```
{
    "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
    }
}
```

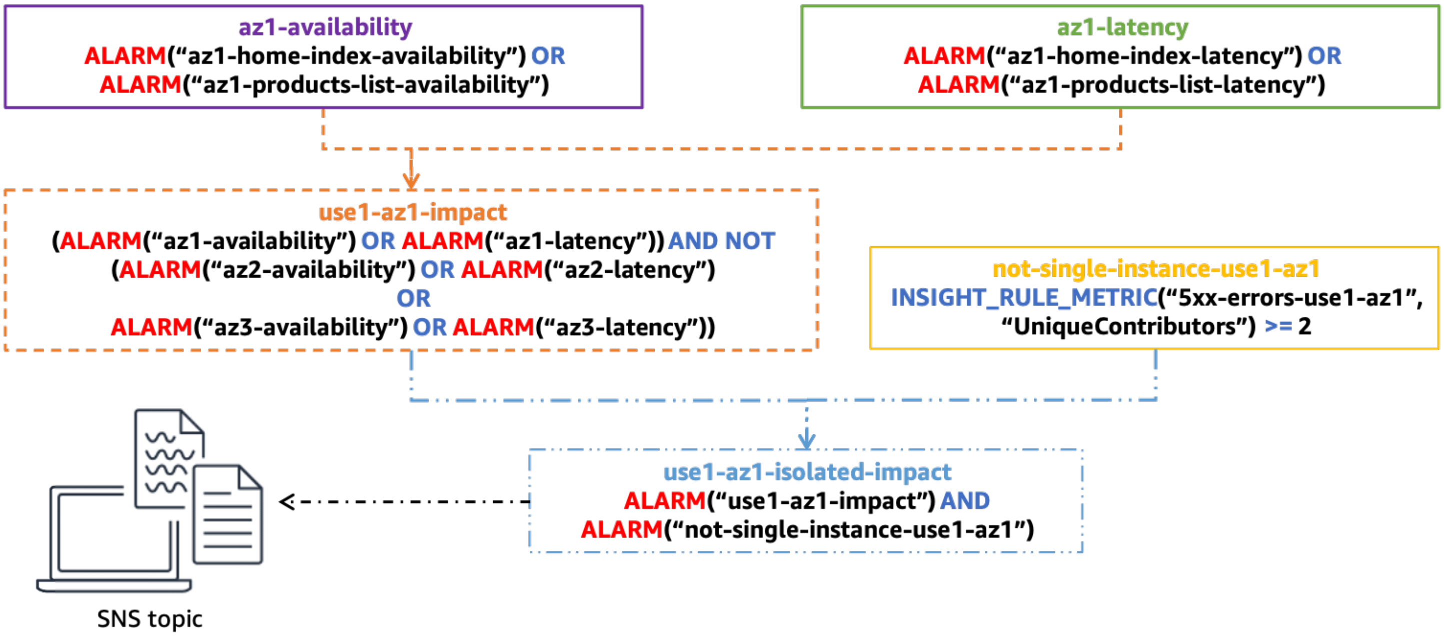
CloudWatch alarms can be created based on these rules as well. You can create alarms based on Contributor Insights rules using metric math and the `INSIGHT_RULE_METRIC` function with the `UniqueContributors` metric. You can also create additional Contributor Insights rules with CloudWatch alarms for metrics like latency or error counts in addition to ones for availability. These alarms can be used with the isolated Availability Zone impact composite alarms to ensure that single instances don't activate the alarm. The metric for the insights rule for `use1-az1` might look like the following:

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

You can define an alarm when this metric is greater than a threshold; for this example, two. It is activated when the unique contributors to 5xx responses goes above that threshold, indicating the impact is originating from more than two instances. The reason this alarm uses a greater-than comparison instead of less-than is to make sure that a zero value for unique contributors doesn't set off the alarm. This tells you that the impact is *not* from a single instance. Adjust this threshold for your individual workload. A general guide is to make this number 5% or more of the total resources in the Availability Zone. More than 5% of your resources being affected shows statistical significance, given a sufficient sample size.

Putting it all together

The following figure shows the complete composite alarm structure for a single Availability Zone:


az1-availability
ALARM("az1-home-index-availability") OR
ALARM("az1-products-list-availability")
az1-latency
ALARM("az1-home-index-latency") OR
ALARM("az1-products-list-latency")
use1-az1-impact
(ALARM("az1-availability") OR ALARM("az1-latency")) AND NOT
(ALARM("az2-availability") OR ALARM("az2-latency")
OR
ALARM("az3-availability") OR ALARM("az3-latency"))
not-single-instance-use1-az1
INSIGHT_RULE_METRIC("5xx-errors-use1-az1",
"UniqueContributors") >= 2
use1-az1-isolated-impact
ALARM("use1-az1-impact") AND
ALARM("not-single-instance-use1-az1")
SNS topic

Complete composite alarm structure for determining single-AZ impact

The final composite alarm, `use1-az1-isolated-impact`, is activated when the composite alarm indicating isolated Availability Zone impact from latency or availability, `use1-az1-aggregate-alarm`, is in `ALARM` state and when the alarm based on the Contributor Insights rule for that same Availability Zone, `not-single-instance-use1-az1`, is also in `ALARM` state (meaning that the impact is more than a single instance). You would create this stack of alarms for each Availability Zone that your workload uses.

You can attach an Amazon Simple Notification Service (Amazon SNS) alert to this final alarm. All of the previous alarms are configured without an action. The alert could notify an operator via email to start manual investigation. It could also initiate automation to evacuate the Availability Zone. However, a word of caution on building automation to respond to these alerts. After an Availability Zone evacuation happens, the result should be that the increased error rates are mitigated and the alarm goes back to an `OK` state. If impact happens in another Availability Zone, it's possible that the automation could evacuate a second or third Availability Zone, potentially removing all of the workload's available capacity. The automation should check to see if an evacuation has already been performed before taking any action. You may also need to scale resources in other Availability Zones before an evacuation is successful.

When you add new controllers or actions to your MVC web app, or a new microservice, or in general, any additional functionality you want to separately monitor, you only need to modify a few alarms in this setup. You will create new availability and latency alarms for that new functionality and then add those to the appropriate Availability Zone aligned availability and latency composite alarms, `az1-latency` and `az1-availability` in the example we've been using here. The remaining composite alarms remain static after they have been configured. This makes onboarding new functionality with this approach a simpler process.

Failure detection using outlier detection

One gap with the previous approach could arise when you see elevated error rates in multiple Availability Zones that are occurring for an *uncorrelated* reason. Imagine a scenario where you have EC2 instances deployed across three Availability Zones and your availability alarm threshold is 99%. Then, a single Availability Zone impairment occurs, isolating many instances and causes availability in that zone to drop to 55%. At the same time, but in a different Availability Zone, a single EC2 instance exhausts all of the storage on its EBS volume, and can no longer write logs files. This causes it to start returning errors, but it still passes the load balancer health checks because those don't trigger a log file to be written. This results in availability dropping to 98% in that Availability Zone. In this case, your single Availability Zone impact alarm wouldn't activate because you are seeing an availability impact in multiple Availability Zones. However, you could still mitigate almost all of the impact by evacuating the impaired Availability Zone.

In some types of workloads, you might experience errors consistently across all Availability Zones where the previous availability metric might not be useful. Take AWS Lambda for example. AWS allows customers to create their own code to run in the Lambda function. To use the service, you have to upload your code in a ZIP file, including dependencies, and define the entry point to the function. But sometimes customers get this part wrong, for example, they might forget a critical dependency in the ZIP file, or mistype the method name in the Lambda function definition. This causes the function to fail to be invoked and results in an error. AWS Lambda sees these kinds of errors all the time, but they're not indicative that anything is necessarily unhealthy. However, something like an Availability Zone impairment might also cause these errors to appear.

To find signal in this kind of noise, you can use outlier detection to determine if there is a statistically significant skew in the number of errors among Availability Zones. Although we see errors across multiple Availability Zones, if there was truly a failure in one of them, we'd expect to see a much higher error rate in that Availability Zone compared to the other ones, or potentially much lower. But how much higher or lower?

One way to do this analysis is by using a chi-squared (χ^2) test to detect statistically significant differences in error rates between Availability Zones (there are many different algorithms for performing outlier detection). Let's look at how the chi-squared test works.

A chi-squared test evaluates the probability that some distribution of results is likely to occur. In this case, we're interested in the distribution of errors across some defined set of AZs. For this example, to make the math easier, consider four Availability Zones.

First, establish the *null hypothesis*, which defines what you believe the default outcome is. In this test, the null hypothesis is that you expect errors to be evenly distributed across each Availability Zone. Then, generate the *alternative hypothesis*, which is that the errors are not evenly distributed indicating an Availability Zone impairment. Now you can test these hypotheses using data from your metrics. For this purpose, you'll sample your metrics from a five-minute window. Suppose you get 1000 published data points in that window, in which you see 100 total errors. You expect that with an even distribution the errors would occur 25% of the time in each of the four Availability Zones. Assume the following table shows what you expected compared to what you actually saw.

Table 1: Expected versus actual errors seen

AZ	Expected	Actual
`use1-az1`	25	20
`use1-az2`	25	20
`use1-az3`	25	25
`use1-az4`	25	35

So, you see that the distribution in reality isn't even. However, you might believe that this occurred due to some level of randomness in the data points you sampled. There's some level of probability that this type of distribution could occur in the sample set and still assume that the null hypothesis is true. This leads to the following question: What is the probability of getting a result at least this extreme? If that probability is below a defined threshold, you reject the null hypothesis. To be *statistically significant*, this probability should be 5% or less.[1]

[1] Craparo, Robert M. (2007). "Significance level". In Salkind, Neil J. Encyclopedia of Measurement and Statistics 3. Thousand Oaks, CA: SAGE Publications. pp. 889–891. ISBN 1-412-91611-9.

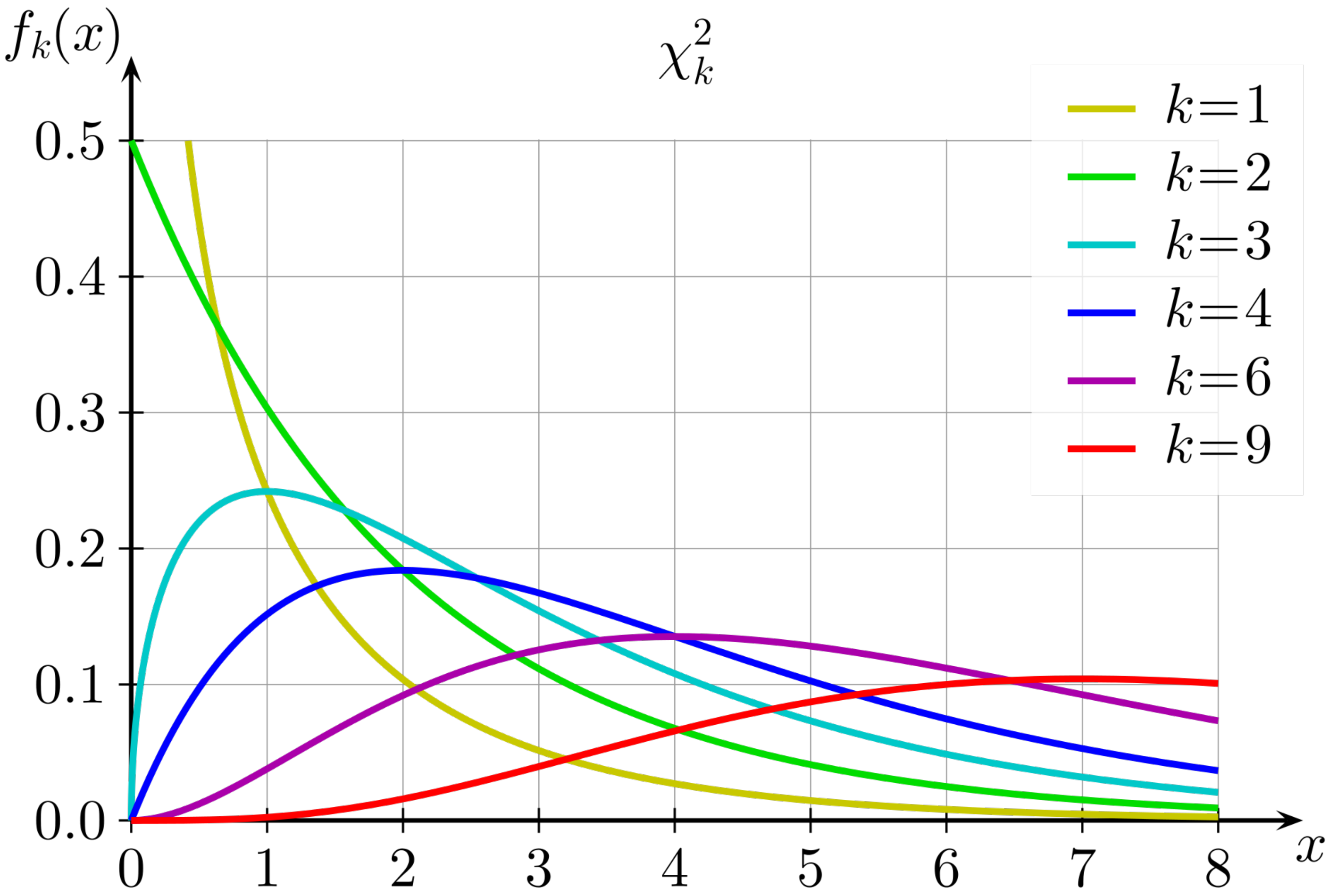
How do you calculate the probability of this outcome? You use the χ^2 statistic that provides very well-studied distributions and can be used to determine the probability of getting a result this extreme or more extreme using this formula.

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

For our example, this results in:

$$\begin{aligned}
\chi^2 &= \frac{(20-25)^2}{25} + \frac{(20-25)^2}{25} + \frac{(25-25)^2}{25} + \frac{(35-25)^2}{25} \\
\chi^2 &= \frac{-5^2}{25} + \frac{-5^2}{25} + \frac{0^2}{25} + \frac{10^2}{25} \\
\chi^2 &= 1 + 1 + 0 + 4 \\
\chi^2 &= 6
\end{aligned} \tag{2}$$

So, what does 6 mean in terms of our probability? You need to look at a chi-squared distribution with the appropriate degree of freedom. The following figure shows several chi-squared distributions for different degrees of freedom.



Chi-squared distributions for different degrees of freedom

The degree of freedom is calculated as one less than the number of choices in the test. In this case, because there are four Availability Zones, the degree of freedom is three. Then, you want to know the area under the curve (the integral) for *x ≥ 6* on the *k = 3* plot. You can also use a pre-calculated table with commonly used values to approximate that value.

Table 2: Chi-squared critical values

Degrees of freedom	**Probability less than the critical value**				
	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

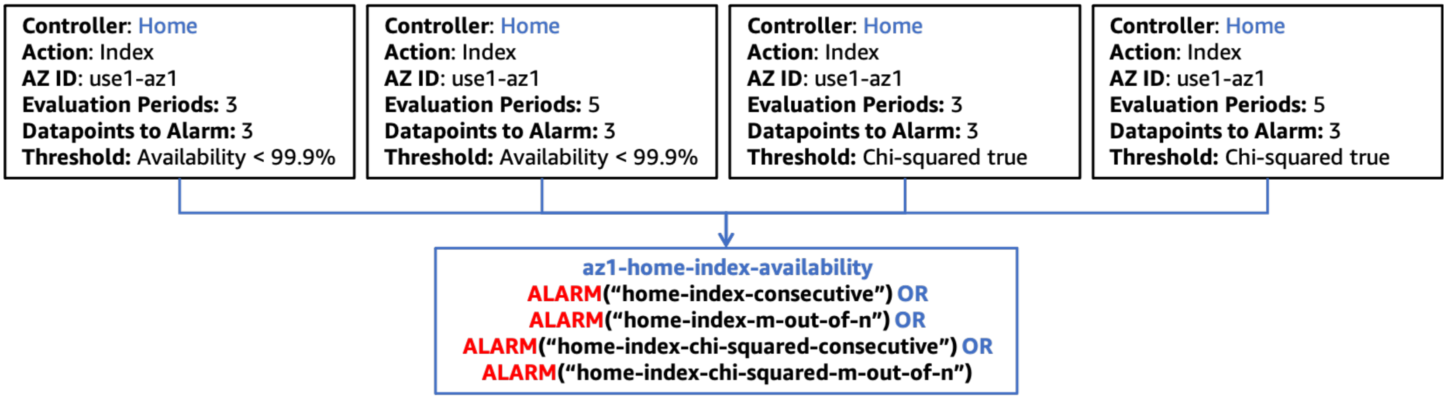
Degrees of freedom	**Probability less than the critical value**				
	0.75	**0.90**	**0.95**	**0.99**	**0.999**
3	4.108	6.251	7.815	11.345	16.266
4	5.385	7.779	9.488	13.277	18.467

For three degrees of freedom, the chi-squared value of six falls between the 0.75 and 0.9 probability columns. What this means is there is a greater than 10% chance of this distribution occurring, which is not less than the 5% threshold. Therefore, you accept the *null hypothesis* and determine there is *not* a statistically significant difference in error rates among the Availability Zones.

Performing a chi-squared statistics test isn't natively supported in CloudWatch metric math, so you'll need collect the applicable error metrics from CloudWatch and run the test in a compute environment like Lambda. You can decide to perform this test at something like an MVC Controller/Action or individual microservice level, or at the Availability Zone level. You'll need to consider whether an Availability Zone impairment would affect each Controller/Action or microservice equally, or whether something like a DNS failure might cause impact in a low throughput service and not in a high throughput service, which could mask the impact when aggregated. In either case, select the appropriate dimensions to create the query. The level of granularity will also impact the resulting CloudWatch alarms you create.

Collect the error count metric for each AZ and Controller/Action in a specified time window. First, calculate the result of the chi-squared test as either true (there was a statistically significant skew) or false (there was wasn't, that is, the null hypothesis holds). If the result is false, publish a 0 data point to your metric stream for chi-squared results for each Availability Zone. If the result is true, publish a 1 data point for the Availability Zone with the errors farthest from the expected value and a 0 for the others (refer to Appendix B – Example chi-squared calculation for sample code that can be used in a Lambda function). You can follow the same approach as the previous availability alarms by using creating a 3 in a row CloudWatch metric alarm and a 3 out of 5 CloudWatch metric alarm based on the data points being produced by the Lambda function. As in the previous examples, this approach can be modified to use more or less data points in a shorter or longer window.

Then, add these alarms to your existing Availability Zone availability alarm for the Controller and Action combination, shown in the following figure.



Integrating the chi-squared statistics test with composite alarms

As mentioned previously, when you onboard new functionality in your workload, you only need to create the appropriate CloudWatch metric alarms that are specific to that new functionality and update the next tier in the composite alarm hierarchy to include those alarms. The rest of the alarm structure remains static.

Failure detection of single instance zonal resources

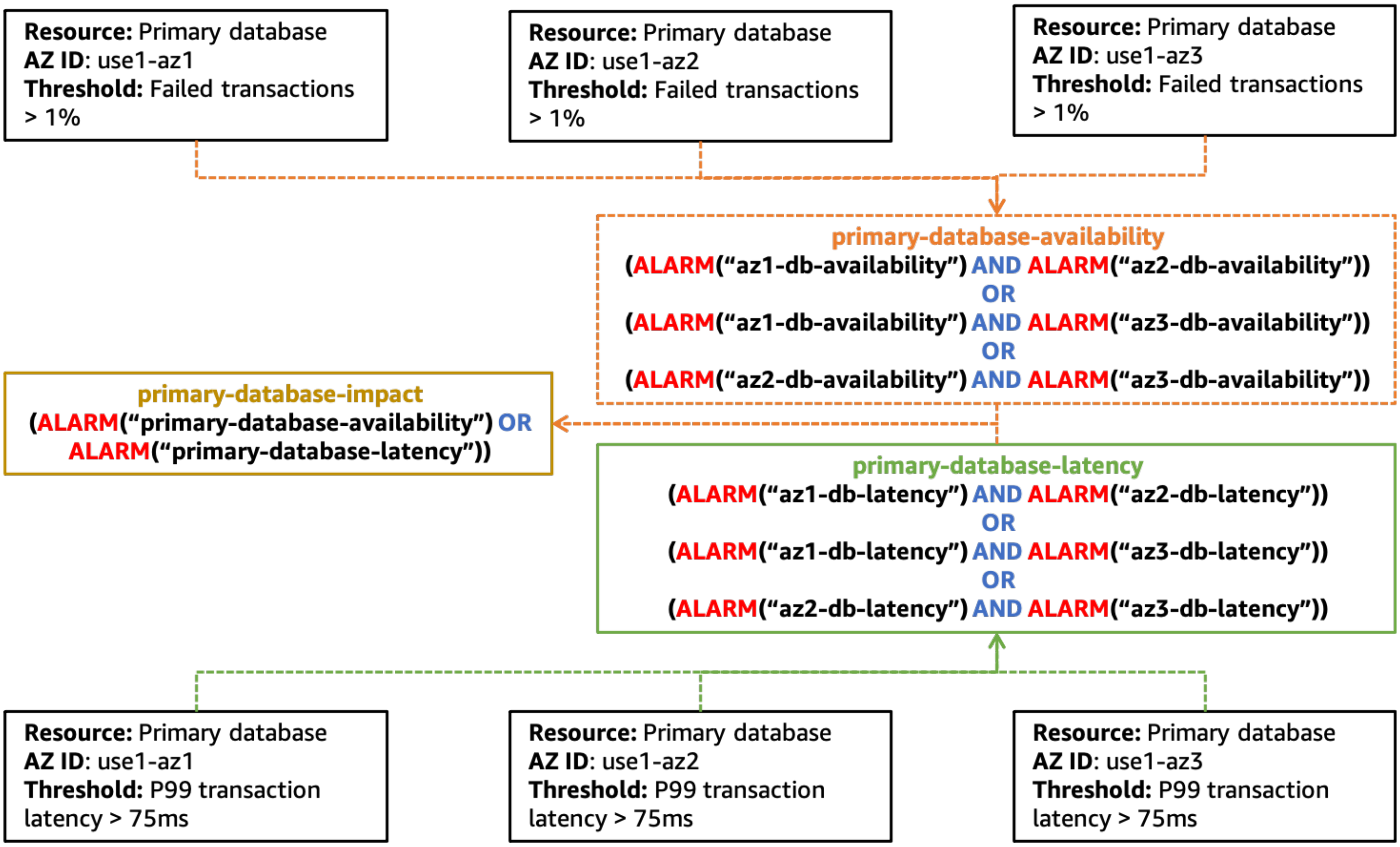
In some cases, you might have a single active instance of a zonal resource, most commonly systems that require a single-writer component such as a relational database (such as Amazon RDS) or a distributed cache (such as Amazon ElastiCache for Redis). If a single Availability Zone impairment affects the Availability Zone that the primary resource is in, it can cause impact to every Availability Zone that accesses the resource. This could cause availability thresholds to be crossed in every Availability Zone, meaning the first approach wouldn't correctly identify the single Availability Zone source of impact. Additionally, you would likely see similar error rates in each Availability Zone, meaning the outlier analysis also wouldn't detect the problem. What this means is that you need to implement additional observability to specifically detect this scenario.

It's likely that the resource you're concerned about will produce its own metrics about its health, but during an Availability Zone impairment that resource might not be able to deliver those metrics. In this scenario, you should create or update alarms to know when you are *flying blind*. If there are important metrics that you already monitor and alarm on, you can configure the alarm to treat the missing data as breaching. This will help you know if the resource stops reporting data, and can be included in the same *in a row* and *m out of n* alarms used previously.

It’s also possible that in some of the metrics that indicate the health of the resource that it publishes a zero valued data point when there is no activity. If the impairment is preventing interactions with the resource, you can’t use the missing data approach for these kinds of metrics. You also probably don’t want to alarm on the value being zero, since there could be legitimate scenarios where that is within normal thresholds. The best approach to detecting this type of problem is with metrics being produced by the resources using this dependency. In this case we want to detect impact in *multiple* Availability Zones using composite alarms. These alarms should use a handful of critical metrics categories related to the resource. A few examples are listed below:

- **Throughput –** The rate of incoming units of work. This could be transactions, reads, writes, and so on.
- **Availability –** Measure the number of successful vs failed units of work.
- **Latency –** Measure multiple percentiles of latency for successful work performed across critical operations.

  Once again, you can create the *in a row* and *m out of n* metric alarms for each metric in each metric category that you want to measure. As before, these can be combined into a composite alarm to determine that this shared resource is the source of impact across Availability Zones. You want to be able to identify impact to more than one Availability Zone with the composite alarms, but the impact does not necessarily need to be *all* Availability Zones. The high-level composite alarm structure for this kind of approach is shown in the following figure.


Resource: Primary database
AZ ID: use1-az1
Threshold: Failed transactions > 1%
Resource: Primary database
AZ ID: use1-az2
Threshold: Failed transactions > 1%
Resource: Primary database
AZ ID: use1-az3
Threshold: Failed transactions > 1%
primary-database-availability
(ALARM("az1-db-availability") AND ALARM("az2-db-availability"))
OR
(ALARM("az1-db-availability") AND ALARM("az3-db-availability"))
OR
(ALARM("az2-db-availability") AND ALARM("az3-db-availability"))
primary-database-impact
(ALARM("primary-database-availability") OR ALARM("primary-database-latency"))
primary-database-latency
(ALARM("az1-db-latency") AND ALARM("az2-db-latency"))
OR
(ALARM("az1-db-latency") AND ALARM("az3-db-latency"))
OR
(ALARM("az2-db-latency") AND ALARM("az3-db-latency"))
Resource: Primary database
AZ ID: use1-az1
Threshold: P99 transaction latency > 75ms
Resource: Primary database
AZ ID: use1-az2
Threshold: P99 transaction latency > 75ms
Resource: Primary database
AZ ID: use1-az3
Threshold: P99 transaction latency > 75ms

An example of creating alarms to detect impact to multiple Availability Zones caused by a single resource

You will notice that this diagram is less prescriptive about what type of metric alarms should be used and the hierarchy of the composite alarms. This is because discovering this kind of problem can be difficult and will require careful attention to the right signals for the shared resource. Those signals may also need to be evaluated in specific ways.

Additionally, you should notice that the `primary-database-impact` alarm is not associated with a specific Availability Zone. This is because the primary database instance can be located in any Availability Zone that it is configured to use, and there's not a CloudWatch metric that specifies where it is. When you see this alarm activate, you should use it as a signal that there may be a problem with the resource and initiate a failover to another Availability Zone, if it hasn't been done automatically. After moving the resource to another Availability Zone, you can wait and see if your isolated Availability Zone alarm is activated, or you can choose to preemptively invoke your Availability Zone evacuation plan.

Summary

This section described three approaches to help identify single Availability Zone impairments. Each approach should be used together to provide a holistic view of your workload's health.

The CloudWatch composite alarm approach allows you to find problems where the skew in availability isn't statistically significant, say availabilities of 98% (the impaired Availability Zone), 100%, and 99.99%, that isn't caused by a single, shared resource.

Outlier detection will help detect single Availability Zone impairments where you have uncorrelated errors happening in multiple Availability Zones that all surpass your alarm threshold.

Finally, identifying degradation of a single instance zonal resource helps discover when an Availability Zone impairment affects a resource that is shared across Availability Zones.

The resulting alarms from each one of these patterns can be combined into a CloudWatch composite alarm hierarchy to discover when single Availability Zone impairments occur and have impact to the availability or latency of your workload.

Availability Zone evacuation patterns

After detecting impact in a single Availability Zone, the next step is to evacuate that Availability Zone. There are two outcomes that evacuation needs to achieve.

First, you want to stop sending work to the impacted Availability Zone. This could mean different things in different architectures. In a request/response workload, this would mean stopping things like HTTP or gRPC requests coming from your customers being sent to the load balancer or other resources in the Availability Zone. In a batch processing or queue processing system, it could mean stopping compute resources from processing work in the impacted Availability Zone. You will also need to prevent resources in the unaffected Availability Zones from interacting with resources in the impacted Availability Zone, for example, an EC2 instance sending traffic to an interface VPC endpoint in the impacted Availability Zone or connecting to the primary instance of a database.

The second outcome is preventing new capacity from being provisioned in the impacted Availability Zone. This is important because new resources, like EC2 instances or containers, being provisioned in the affected Availability Zone are likely to see the same impact as existing resources. Additionally, because the first outcome prevents work from being sent to them, they cannot absorb the load they were provisioned to handle. This leads to increased load on the existing resources, which can ultimately lead to *brown out* or total unavailability of the workload. There are several auto scaling services available in AWS where this is applicable: Amazon EC2 Auto Scaling, Application Auto Scaling, and AWS Auto Scaling. Additionally, services like Amazon ECS, Amazon EKS, and AWS Batch may schedule work on hosts across Availability Zones in a VPC as part of their normal operation.

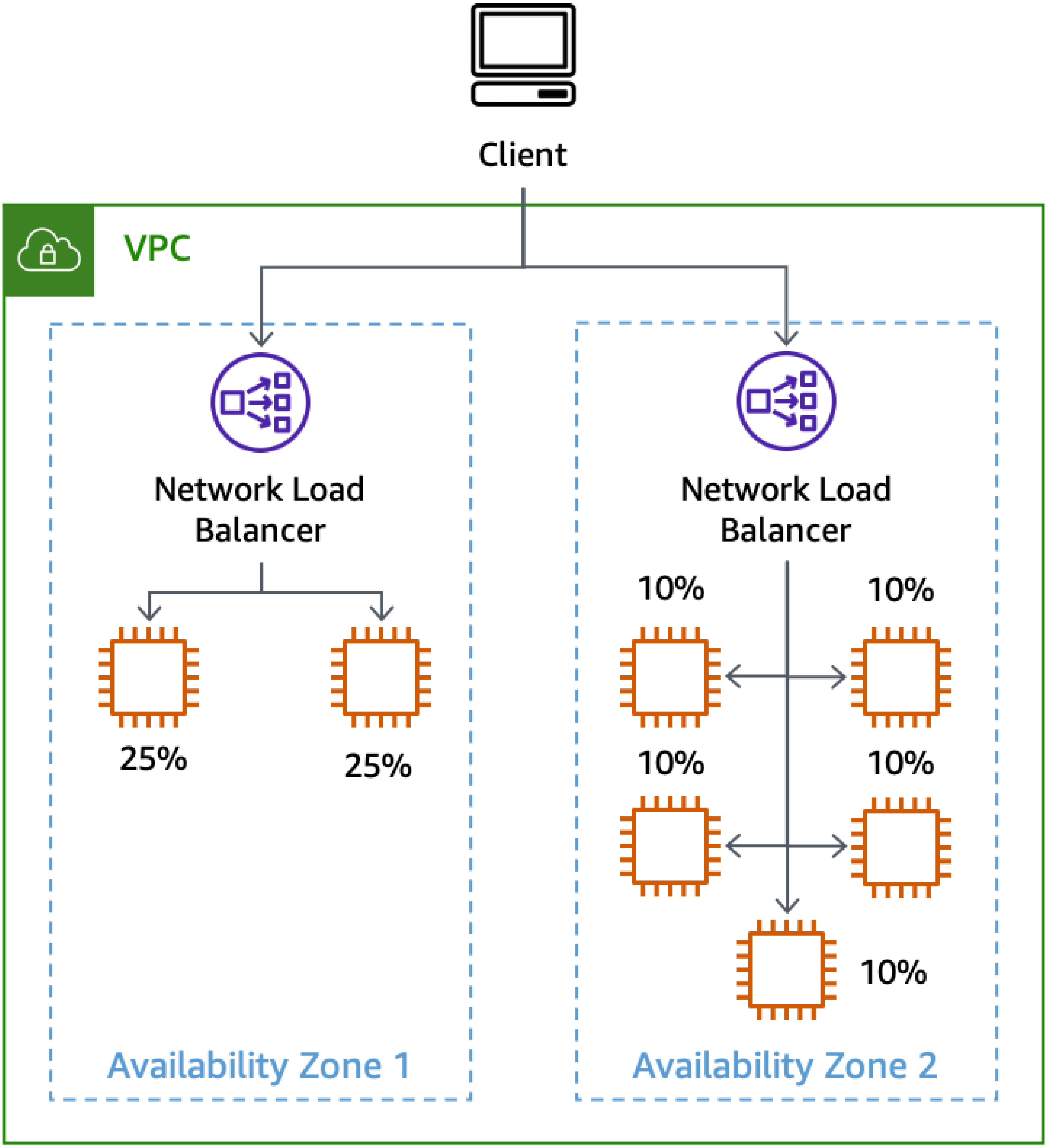
Topics

Availability Zone independence

To achieve the first outcome, to stop sending work to the impacted Availability Zone, evacuation requires that you implement Availability Zone Independence (AZI), also sometimes called

Availability Zone Affinity. This architectural pattern isolates resources inside an Availability Zone and prevents interaction among resources in different Availability Zones except where absolutely required, such as connecting to a primary database instance in a different Availability Zone.

In a request/response type workload, implementing AZI requires you to disable cross-zone load balancing for Application Load Balancers (ALB), Classic Load Balancers (CLB), and Network Load Balancers (NLB) (cross-zone load balancing is disabled by default for NLBs). Disabling cross-zone load balancing has a few tradeoffs. When you disable cross-zone load balancing, traffic is evenly split between each Availability Zone regardless of how many instances are in each one. If you have unbalanced resources or Auto Scaling groups, this could put additional load on resources in an Availability Zone that has fewer resources than others. This is shown in the following figure where two instances in Availability Zone 1 are each receiving 25% of the load and the five instances in Availability Zone 2 are each receiving 10% of the load.



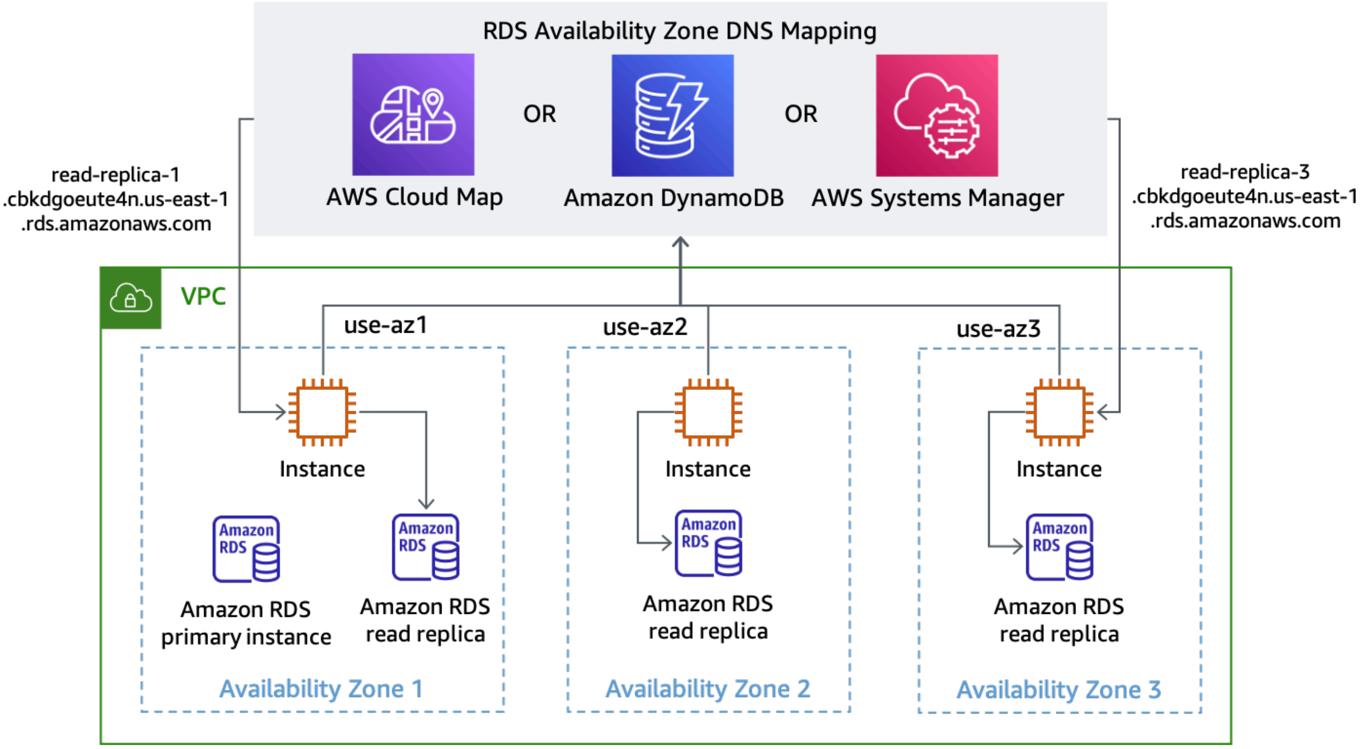
The effect of disabling cross-zone load balancing with unbalanced instances

Other zonal services that you use will also need to be implemented using AZI patterns to support effective Availability Zone evacuation. For example, interface VPC endpoints provide specific DNS names for each Availability Zone the interface endpoint is made available in.

One challenge with implementing AZI is with databases, especially because most relational databases only support a single primary writer at any time. When communicating with the primary instance, you may need to cross an Availability Zone boundary. Many AWS database services support a user-defined Multi-AZ configuration and have a built-in multi-AZ failover feature, such as Amazon RDS or Amazon Aurora. In many failure scenarios, the service can detect the impact and automatically failover the database to a different Availability Zone when a problem occurs. However, during a gray failure, the service may not detect the impact that is affecting your workload, or the impact may not be related to the database at all. In these cases, once you detect impact in an Availability Zone, you can manually invoke a failover to move the primary database. This allows you to effectively react to a single Availability Zone impairment.

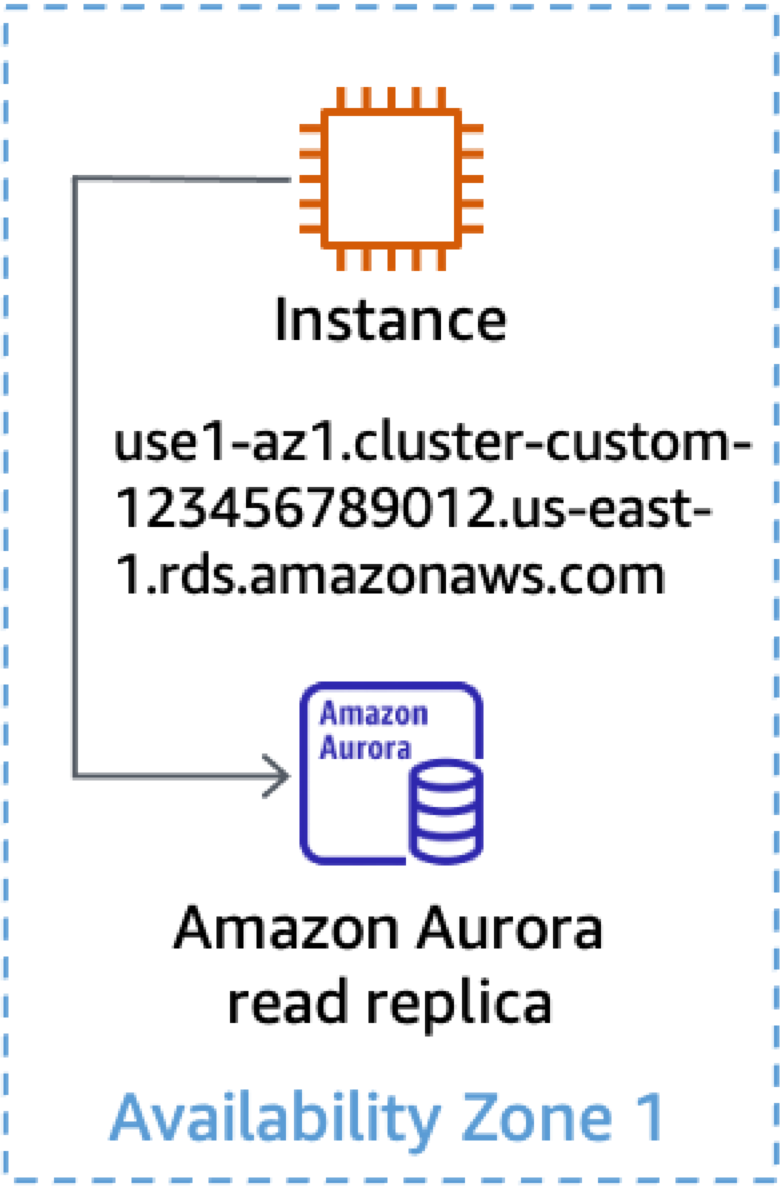
If you are using read replicas with those databases, you may also want to implement AZI for those because you cannot failover a read replica to a different Availability Zone like you can the primary database. If you have a single read replica in Availability Zone 1, and instances across three Availability Zones are configured to use it, an impairment impacting Availability Zone 1 will also impact operations in the other two Availability Zones. That’s the impact you want to prevent.

For RDS instances, you receive a DNS endpoint to access the replica in a specific Availability Zone. To achieve AZI, you would need a read replica per Availability Zone and a way for your application to know which replica endpoint to use for the Availability Zone it’s in. One approach you can take is to use the Availability Zone ID as part of the database identifier, something like `use1-az1-read-replica.cbkdgoeute4n.us-east-1.rds.amazonaws.com`. You can also do this using service discovery (such as with AWS Cloud Map) or looking up a simple map stored in AWS Systems Manager Parameter Store or a DynamoDB table. This concept is shown in the following figure.



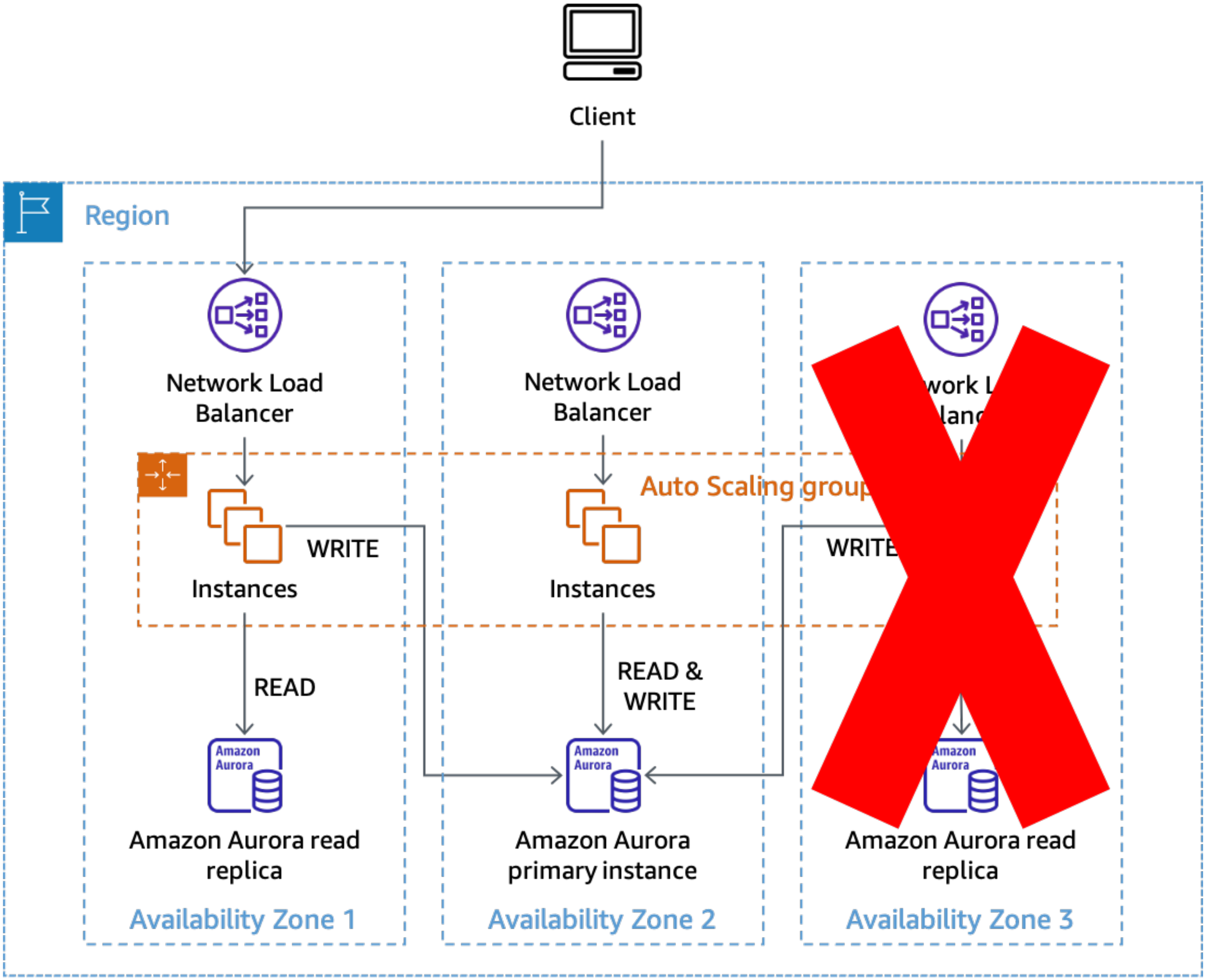
Discovering RDS endpoint DNS names for each Availability Zone

Amazon Aurora's default configuration is to provide a single reader endpoint that load balances requests across available read replicas. In order to implement AZI using Aurora, you can use a custom endpoint for each read replica using the ANY type (so you can promote a read replica if required). Name the custom endpoint based on the Availability Zone ID where the replica is deployed. Then, you can use the DNS name provided by the custom endpoint to connect to a specific read replica in a specific Availability Zone, which is shown in the following figure.


Instance
use1-az1.cluster-custom-123456789012.us-east-1.rds.amazonaws.com
Amazon Aurora
Amazon Aurora read replica
Availability Zone 1

Using a custom endpoint for an Aurora read replica

When your system is architected this way, it makes Availability Zone evacuation a much simpler task. For example, in the following figure when there is an impairment affecting Availability Zone 3, both read and write operations in Availability Zones 1 and 2 are not affected.



Using AZI to prevent impact with Amazon Aurora read replicas

Alternatively, if Availability Zone 2 was impacted, read operations would still succeed in Availability Zones 1 and 3. Then, if Amazon Aurora hasn't automatically failed over the primary database, you can manually invoke a failover to a different Availability Zone to restore the capability for processing writes. This approach prevents needing to make any configuration changes in your database connections when you need to evacuate an Availability Zone. Minimizing the required changes and keeping the process as simple as possible will make it more reliable.

Control planes and data planes

Before we get to the actual patterns you can use to perform an Availability Zone evacuation, we need to discuss the concepts of control planes and data planes. AWS makes a distinction between control planes and data planes in our services. Control planes are the machinery involved in making changes to a system—adding resources, deleting resources, modifying resources—and getting those changes propagated to wherever they need to go to take effect, such as updating a network configuration for an ALB or creating an AWS Lambda function.

Data planes are the primary function of those resources, things such as the running EC2 instance, or getting items from or putting items into an Amazon DynamoDB table. For a more detailed discussion of control planes and data planes, refer to Static stability using Availability Zones and AWS Fault Isolation Boundaries.

For the purposes of this document, consider that control planes tend to have more moving parts and dependencies than data planes. This makes it statistically more likely that the control plane becomes impaired compared to the data plane. This is especially relevant for services that provide AZI, such as Amazon EC2 and EBS, because parts of those services have control planes that are also zonally independent and can be impacted during a single-AZ event.

While control plane actions can be used to perform AZ evacuation, based on the previous information, they may have a lower probability of success, especially during a failure event. To increase the probability of successfully mitigating impact, you can use two different patterns. The first pattern relies only on data plane actions to initially mitigate impact by preventing work from being routed to or stop work from being done in the impacted Availability Zone. Then, the second pattern can be attempted to update the configuration of resources with control plane actions to both prevent capacity from being provisioned in the impacted Availability Zone as well as stop inter-Availability Zone communication with that Availability Zone.

The recovery patterns discussed in this section are *big red buttons*. They are the mechanisms you use to take large-scale action, quickly, akin to pulling an Andon cord on an assembly line. They assume that the workloads have already attempted strategies such as retry with exponential backoff with jitter in their code to overcome transient errors. This means that when isolated Availability Zone impact is detected, its effects on availability or latency are severe enough to require evacuating the Availability Zone to effectively mitigate.

Data plane-controlled evacuation

There are several solutions that you can implement to perform an Availability Zone evacuation using data plane-only actions. This section will describe three of them and the use cases where you may want to pick one over the other.

When using any of these solutions, you need to ensure you have sufficient capacity in the remaining Availability Zones to handle the load of the Availability Zone you are shifting away from. The most resilient was to do this is by having the required capacity pre-provisioned in each Availability Zone. If you are using three Availability Zones, you would have 50% of the required capacity to handle your peak load deployed in each one, so that the loss of a single Availability Zone would still leave you 100% of your required capacity without having to rely on a control plane to provision more.

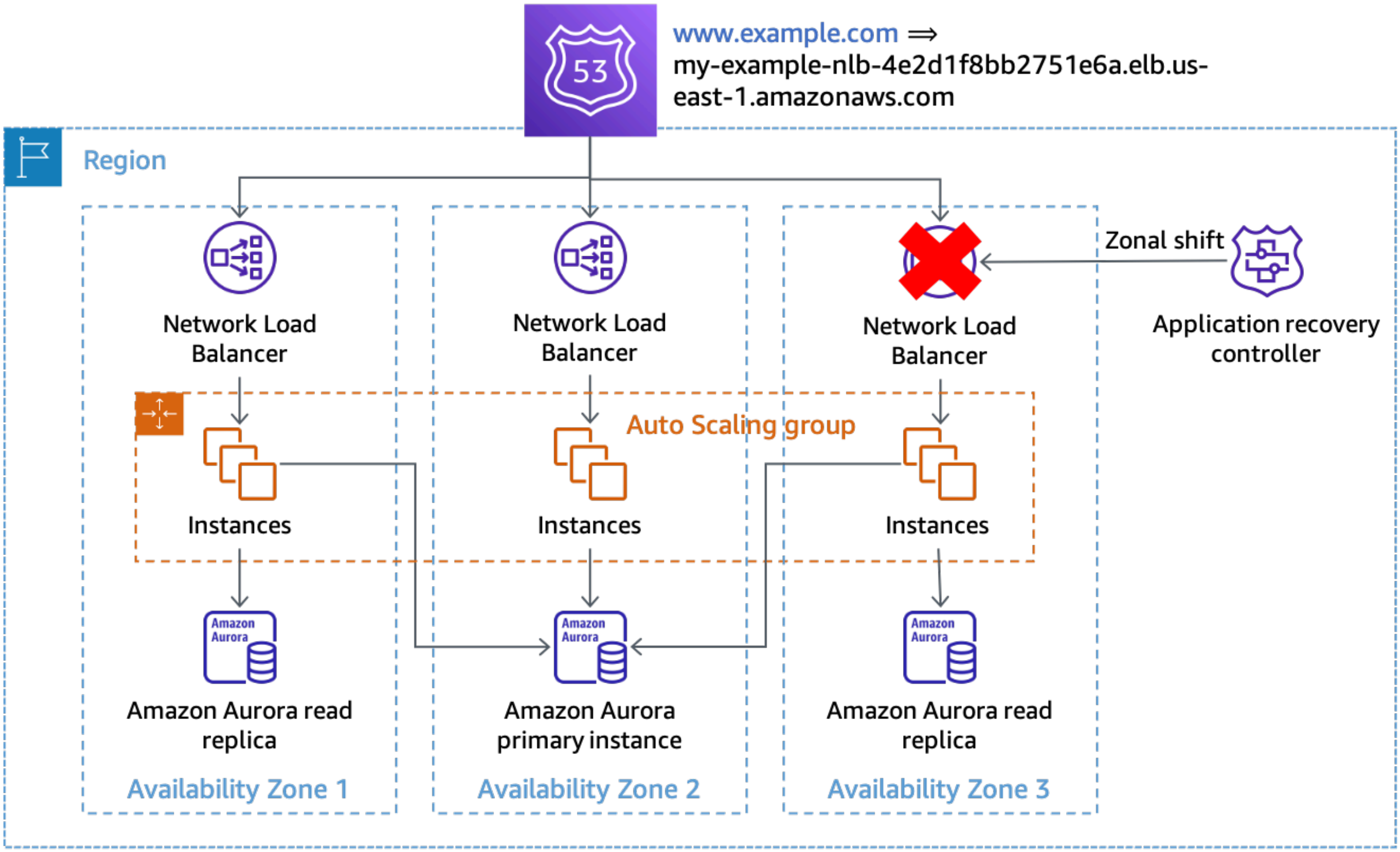
Additionally, if you are using EC2 Auto Scaling, ensure your Auto Scaling group (ASG) doesn't scale in during the shift, so that when the shift ends, you still have sufficient capacity in the group to handle your customer traffic. You can do this by ensuring that your ASG's minimum desired capacity can handle your current customer load. You can also help ensure that your ASG doesn't inadvertently scale in by using averages in your metrics as opposed to outlier percentile metrics like P90 or P99.

During a shift, the resources no longer serving traffic should have very low utilization, but the other resources will increase their utilization with the new traffic, keeping the average fairly consistent, which would prevent a scale-in action. Finally, you can also use target group health settings for ALB and NLB to specify DNS failover with either a percentage or count of healthy hosts. This prevents traffic from being routed to an Availability Zone that does not have enough healthy hosts.

Zonal Shift in Route 53 Application Recovery Controller (ARC)

The first solution for Availability Zone evacuation uses zonal shift in Route 53 ARC. This solution can be used for request/response workloads that use an NLB or ALB as the ingress point for customer traffic.

When you detect that an Availability Zone has become impaired, you can initiate a zonal shift with Route 53 ARC. Once this operation completes and existing cached DNS responses expire, all new requests are only routed to resources in the remaining Availability Zones. The following figure shows how zonal shift works. In the following figure we have a Route 53 alias record for `www.example.com` that points to `my-example-nlb-4e2d1f8bb2751e6a.elb.us-east-1.amazonaws.com`. The zonal shift is performed for Availability Zone 3.



Zonal shift

In the example, if the primary database instance is not in Availability Zone 3, then performing the zonal shift is the only action required to achieve the first outcome for evacuation, preventing work from being processed in the impacted Availability Zone. If the primary node was in Availability Zone 3, then you could perform a manually initiated failover (which does rely on the Amazon RDS control plane) in coordination with the zonal shift, if Amazon RDS did not already failover automatically. This will be true for all of the data plane-controlled solutions in this section.

You should initiate the zonal shift using CLI commands or the API in order to minimize dependencies required to start the evacuation. The simpler the evacuation process, the more reliable it will be. The specific commands can be stored in a local runbook that on-call engineers can easily access. Zonal shift is the most preferred and simplest solution for evacuating an Availability Zone.

Route 53 ARC

The second solution uses the capabilities of Route 53 ARC to manually specify the health of specific DNS records. This solution has the benefit of using the highly available Route 53 ARC cluster data plane, making it resilient to the impairment of up to two different AWS Regions. It has the tradeoff of additional cost and it requires some additional configuration of DNS records. To implement this pattern, you need to create alias records for the Availability Zone-specific DNS names provided by the load balancer (ALB or NLB). This is shown in the following table.

Table 3: Route 53 alias records configured for the load balancer's zonal DNS names

Routing Policy: weighted	**Routing Policy:** weighted	**Routing Policy:** weighted
Name: www.example.com	**Name:** www.example.com	**Name:** www.example.com
Type: A (alias)	**Type:** A (alias)	**Type:** A (alias)
Value: us-east-1b.load-balancer-name.elb.us-east-1.amazonaws.com	**Value:** us-east-1a.load-balancer-name.elb.us-east-1.amazonaws.com	**Value:** us-east-1c.load-balancer-name.elb.us-east-1.amazonaws.com
Weight: 100	**Weight:** 100	**Weight:** 100
Evaluate Target Health: true	**Evaluate Target Health:** true	**Evaluate Target Health:** true

For each of these DNS records, you would configure a Route 53 health check that is associated with a Route 53 ARC routing control. When you want to initiate an Availability Zone evacuation, set the routing control state to Off. AWS recommends you do this using the CLI or API in order to minimize the dependencies required to start the Availability Zone evacuation. As a best practice, you should keep a local copy of the Route 53 ARC cluster endpoints so you don't need to retrieve those from the ARC control plane when you need to perform an evacuation.

To minimize cost when using this approach, you can create a single Route 53 ARC cluster and health checks in a single AWS account and share the health checks with other AWS accounts in your organization. When you take this approach, you should use the Availability Zone ID (AZ-ID) (for example, use1-az1) instead of the Availability Zone name (for example, us-east-1a) for your routing controls. Because AWS maps the physical Availability Zone randomly to the Availability Zone names for each AWS account, using the AZ-ID provides a consistent way to refer to the same physical locations. When you initiate an Availability Zone evacuation, say for use1-az2, the

Route 53 record sets in each AWS account should ensure they use the AZ-ID mapping to configure the right health check for each NLB records.

For example, let's say we have a Route 53 health check associated with a Route 53 ARC routing control for `use1-az2`, with an ID of `0385ed2d-d65c-4f63-a19b-2412a31ef431`. If in a different AWS account that wants to use this health check, `us-east-1c` was mapped to `use1-az2`, you would need to use the `use1-az2` health check for the record `us-east-1c.load-balancer-name.elb.us-east-1.amazonaws.com`. You would use the health check ID `0385ed2d-d65c-4f63-a19b-2412a31ef431` with that resource record set.

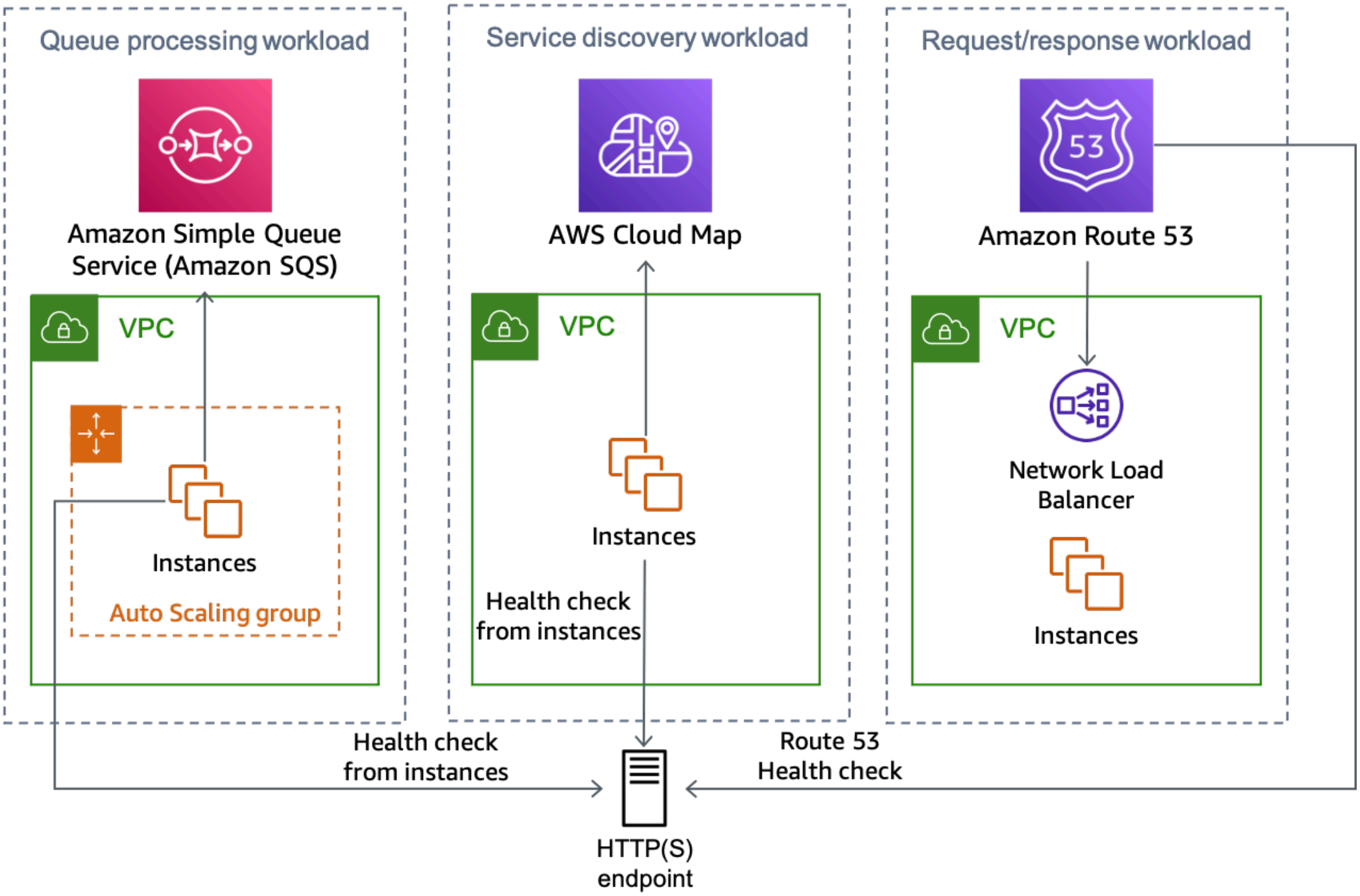
Using a self-managed HTTP endpoint

You can also implement this solution by managing your own HTTP endpoint that indicates the status of a particular Availability Zone. It allows you to manually specify when an Availability Zone is unhealthy based on the response from the HTTP endpoint. This solution costs less than using Route 53 ARC, but is more expensive than zonal shift and requires managing additional infrastructure. It has the benefit of being much more flexible for different scenarios.

The pattern can be used with NLB or ALB architectures and Route 53 health checks. It can also be used in non-load balanced architectures, like service discovery or queue processing systems where worker nodes perform their own health checks. In those scenarios, the hosts can use a background thread where they periodically make a request to the HTTP endpoint with their AZ-ID (refer to Appendix A – Getting the Availability Zone ID for details on how find this) and receive back a response about the health of the Availability Zone.

If the Availability Zone has been declared to be unhealthy, they have multiple options on how to respond. They may choose to fail an external health check from sources such as ELB, Route 53, or custom health checks in service discovery architectures so that they appear unhealthy to those services. They can also immediately respond with an error should they receive a request, allowing the client to backoff and retry. In event-driven architectures, nodes can intentionally fail to process work, like intentionally returning an SQS message to the queue. In work router architectures where a central service schedules work on specific hosts you can also use this pattern. The router can check the status of an Availability Zone before selecting a worker, endpoint, or cell. In service discovery architectures that use AWS Cloud Map, you can discover endpoints by providing a filter in your request, such as an AZ-ID.

The following figure shows how this approach can be used for multiple types of workloads.


Queue processing workload
Amazon Simple Queue Service (Amazon SQS)
VPC
Instances
Auto Scaling group
Service discovery workload
AWS Cloud Map
VPC
Instances
Health check from instances
Request/response workload
Amazon Route 53
VPC
Network Load Balancer
Instances
Health check from instances
Route 53 Health check
HTTP(S) endpoint

Multiple workload types can all use the HTTP endpoint solution

There are multiple ways to implement the HTTP endpoint approach, two of them are outlined next.

Using Amazon S3

This pattern was originally presented in this blog post for multi-Region disaster recovery. You can use the same pattern for Availability Zone evacuation.

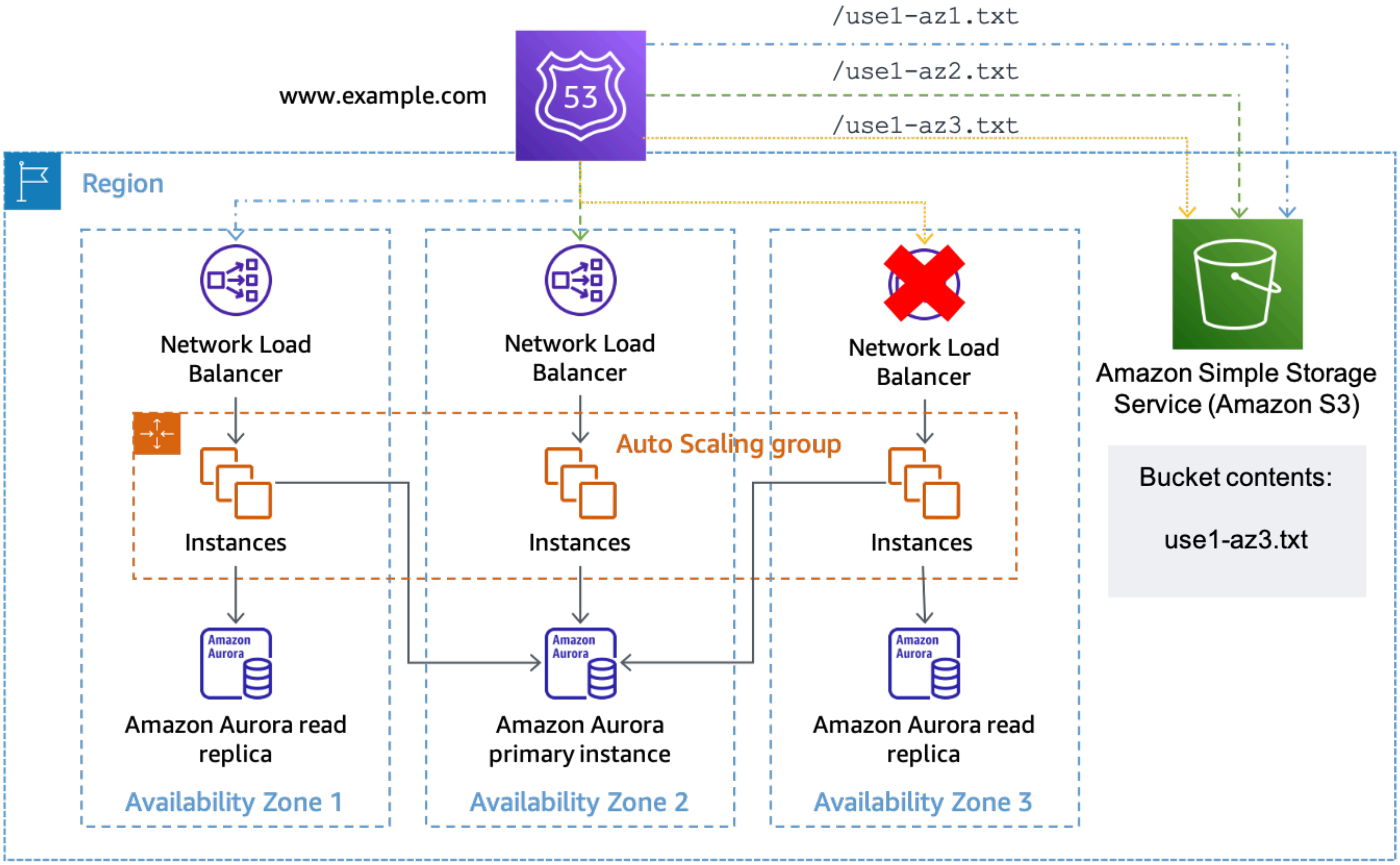
In this scenario you would create Route 53 DNS resource record sets for each zonal DNS record just like the *Route 53 ARC* scenario above as well as associated health checks. However, for this implementation, instead of associating the health checks with Route 53 ARC routing controls, they are configured to use an HTTP endpoint and are inverted to safeguard against an impairment in Amazon S3 accidentally triggering an evacuation. The health check is considered *healthy* when the object is absent and *unhealthy* when the object is present. This setup is shown in the following table.

Table 4: DNS record configuration for using Route 53 health checks per Availability Zone

Health check type: monitor an endpoint **Protocol:** HTTPS **ID**: dddd-4444 **URL**: https://*bucket name* .s3.us-east-1.amazonaws.com/use1-az1.txt	**Health check type**: monitor an endpoint **Protocol**: HTTPS **ID**: eeee-5555 **URL**: https://*bucket name* .s3.us-east-1.amazonaws.com/use1-az3.txt	**Health check type**: monitor an endpoint **Protocol**: HTTPS **ID**: ffff-6666 **URL**: https://*bucket name* .s3.us-east-1.amazonaws.com/use1-az2.txt	←	**Health checks**
↑	↑	↑		
Routing Policy: weighted **Name**: www.example.com **Type**: A (alias) **Value**: us-east-1b.load-balancer-name.elb.us-east-1.amazonaws.com **Weight**: 100	**Routing Policy:** weighted **Name:** www.example.com **Type:** A (alias) **Value:** us-east-1a.load-balancer-name.elb.us-east-1.amazonaws.com **Weight:** 100	**Routing Policy:** weighted **Name:** www.example.com **Type:** A (alias) **Value:** us-east-1c.load-balancer-name.elb.us-east-1.amazonaws.com **Weight:** 100	←	**Top level, evenly weighted alias A records point to NLB AZ specific endpoints**

Evaluate Target Health: `true`	**Evaluate Target Health:** `true`	**Evaluate Target Health:** `true`		

Let's assume that the Availability Zone `us-east-1a` is mapped to `use1-az3` in the account where we have a workload where we want to perform an Availability Zone evacuation. For the resource record set created for `us-east-1a.load-balancer-name.elb.us-east-1.amazonaws.com` would associate a health check that tests the URL `https://bucket-name.s3.us-east-1.amazonaws.com/use1-az3.txt`. When you want to initiate an Availability Zone evacuation for `use1-az3`, upload a file named `use1-az3.txt` to the bucket using the CLI or API. The file doesn't need to contain any content, but it does need to be public so that the Route 53 health check can access it. The following figure demonstrates this implementation being used to evacuate `use1-az3`.



Using Amazon S3 as the target for a Route 53 health check

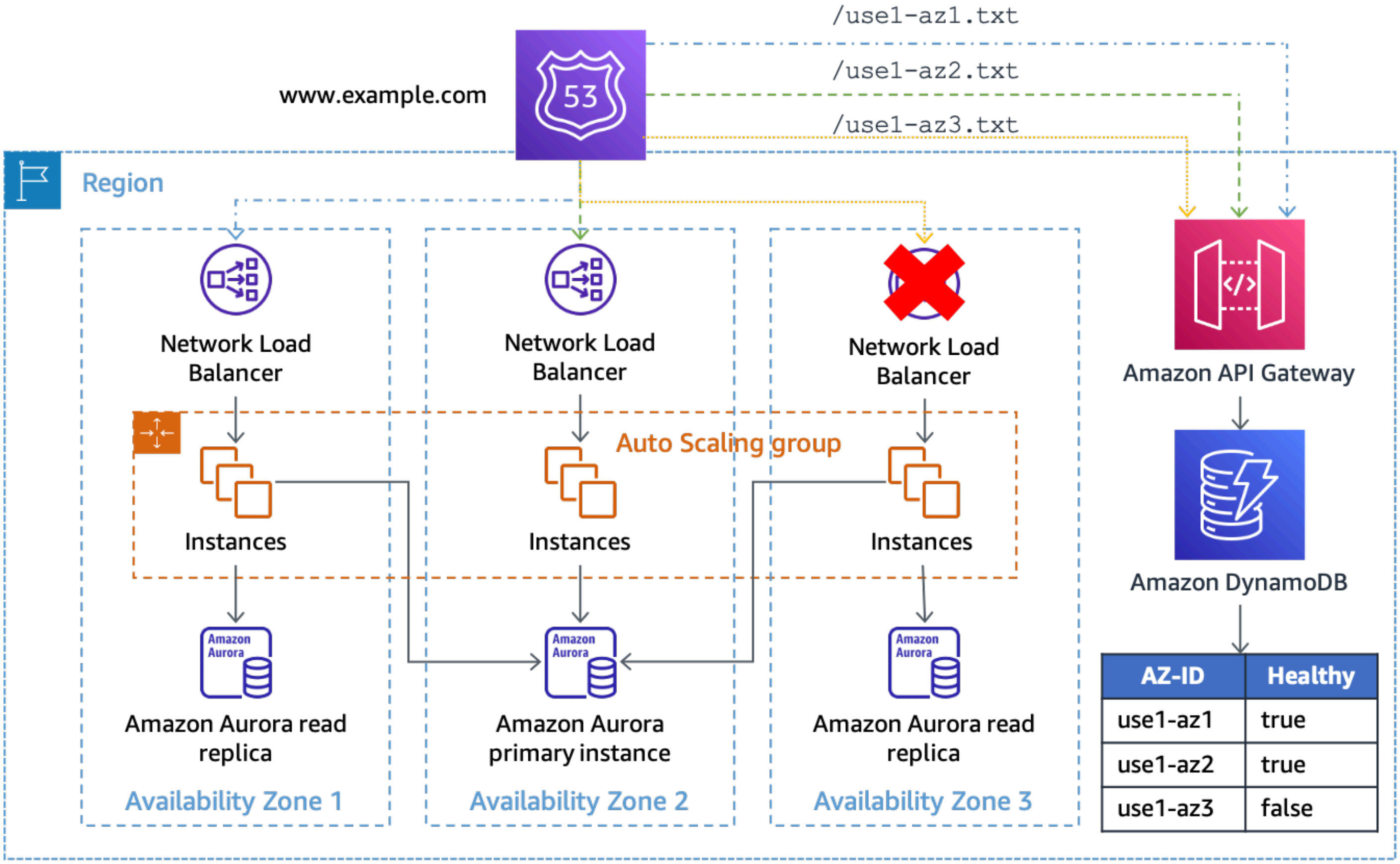
Using API Gateway and DynamoDB

The second implementation of this pattern uses an Amazon API Gateway REST API. The API is configured with a service integration to Amazon DynamoDB where the status for each in-use Availability Zone is stored. This implementation is more flexible than the Amazon S3 approach, but requires building, operating, and monitoring more infrastructure. It can also both be used with Route 53 health checks as well as health checks performed by individual hosts.

If you are using this solution with an NLB or ALB architecture, set up your DNS records in the same way as the Amazon S3 example above, except change the health check path to use the API Gateway endpoint and provide the `AZ-ID` in the URL path. For example, if the API Gateway is configured with a custom domain of `az-status.example.com`, the full request for `use1-az1` would look like `https://az-status.example.com/status/use1-az1`. When you want to initiate an Availability Zone evacuation, you can create or update a DynamoDB item using the CLI or API. The item uses the `AZ-ID` as its primary key and then has a Boolean attribute called `Healthy` which is used indicate how API Gateway responds. The following is example code used in the API Gateway configuration to make this determination.

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

If the attribute is `true` (or isn't present), API Gateway responds to the health check with an HTTP 200, if it is false, it responds with an HTTP 500. This implementation is shown in the following figure.



AZ-ID	Healthy
use1-az1	true
use1-az2	true
use1-az3	false

Using API Gateway and DynamoDB as the target of Route 53 health checks

In this solution you need to use API Gateway in front of DynamoDB so that you can make the endpoint publicly accessible as well as manipulate the request URL into a `GetItem` request for DynamoDB. The solution also provides flexibility if you want to include additional data in the request. For example, if you wanted to create more granular statuses, like per application, you can configure the health check URL to provide an application ID in the path or query string that is also matched against the DynamoDB item.

The Availability Zone status endpoint can be deployed centrally so that multiple health check resources across AWS accounts can all use the same consistent view of Availability Zone health (ensuring that your API Gateway REST API and DynamoDB table are scaled to handle the load) and eliminates the need to share Route 53 health checks.

The solution could also be scaled across multiple AWS Regions using an Amazon DynamoDB global table and a copy of the API Gateway REST API in each Region. This prevents this solution from having a dependency on a single Region and increases its availability. You could deploy the solution across three or five Regions and query each one for Availability Zone health, using the result of the majority of the endpoints to ensure quorum. This allows for eventually consistent replication of updates across the global table as well as mitigates impairments that may prevent an endpoint from responding. For example, if you are using five Regions, and three endpoints report an Availability Zone as unhealthy, one endpoint reports the Availability Zone as healthy, and one endpoint does not respond, you would choose to treat the Availability Zone as unhealthy. You could also create a Route 53 calculated health check using an *m of n* calculation to perform this logic to determine Availability Zone health.

If you were building a solution for individual hosts to use as a mechanism to determine the health of their AZ, as an alternative, instead of providing a pull mechanism for health checks, you can use push notifications. One way to do this is with an SNS topic that your consumers subscribe to. When you want to trigger the circuit breaker, publish a message to the SNS topic that indicates which Availability Zone is impaired. This approach makes tradeoffs with the former. It removes the need to create and operate the API Gateway infrastructure and perform capacity management. It can also potentially provide faster convergence of the Availability Zone state. However, it removes the ability to perform ad hoc queries and relies on the SNS delivery retry policy to ensure each endpoint receives the notification. It also requires each workload or service to build a way to receive the SNS notification and take action on it.

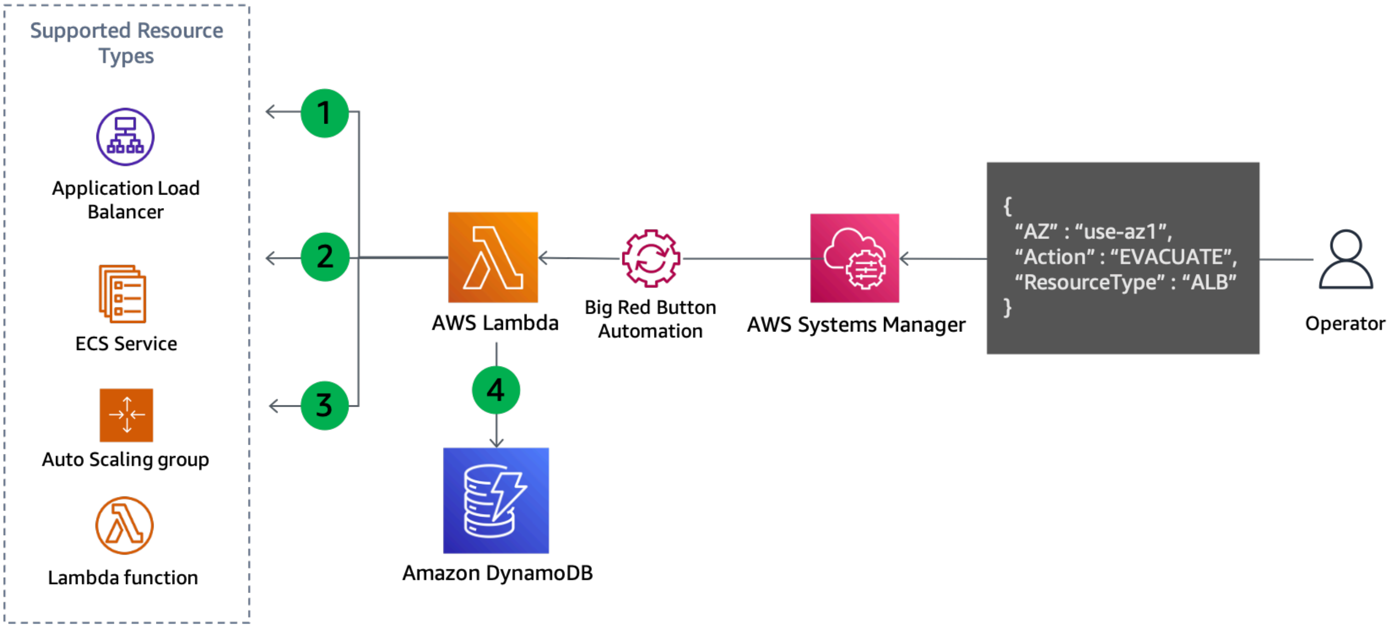
For example, each new EC2 instance or container that is launched will need to subscribe to the topic with an HTTP endpoint during its bootstrap. Then, each instance needs to implement software that listens on this endpoint where the notification is delivered. Additionally, if the instance is impacted by the event, it may not receive the push notification and continue to do work. Whereas, with a pull notification, the instance will know if its pull request fails and can choose what action to take in response.

A second way to send push notifications is with long-lived WebSocket connections. Amazon API Gateway can be used to provide a WebSocket API that consumers can connect to and receive a message when sent by the backend. With a WebSocket, instances can both do periodic pulls to ensure their connection is healthy and also receive low-latency push notifications.

Control plane-controlled evacuation

The first pattern uses data plane operations to prevent performing work in an impacted Availability Zone to mitigate the impact of an event. However, you may be using an architecture that doesn’t use load balancers or where configuring a per-host health check isn’t feasible. Or, you may want to prevent new capacity from being deployed into the impacted Availability Zone through Auto Scaling or normal work scheduling.

To address both situations, control plane actions are required to update the configuration of the resource. The pattern will work for any service whose network configuration can be updated, for example, EC2 Auto Scaling, Amazon ECS, Lambda, and more. It requires writing code for each service, but the business logic follows a standard pattern. The code should be executed locally by an operator responding to the event in order to minimize the dependencies required. The basic flow of the script logic is shown in the following figure.

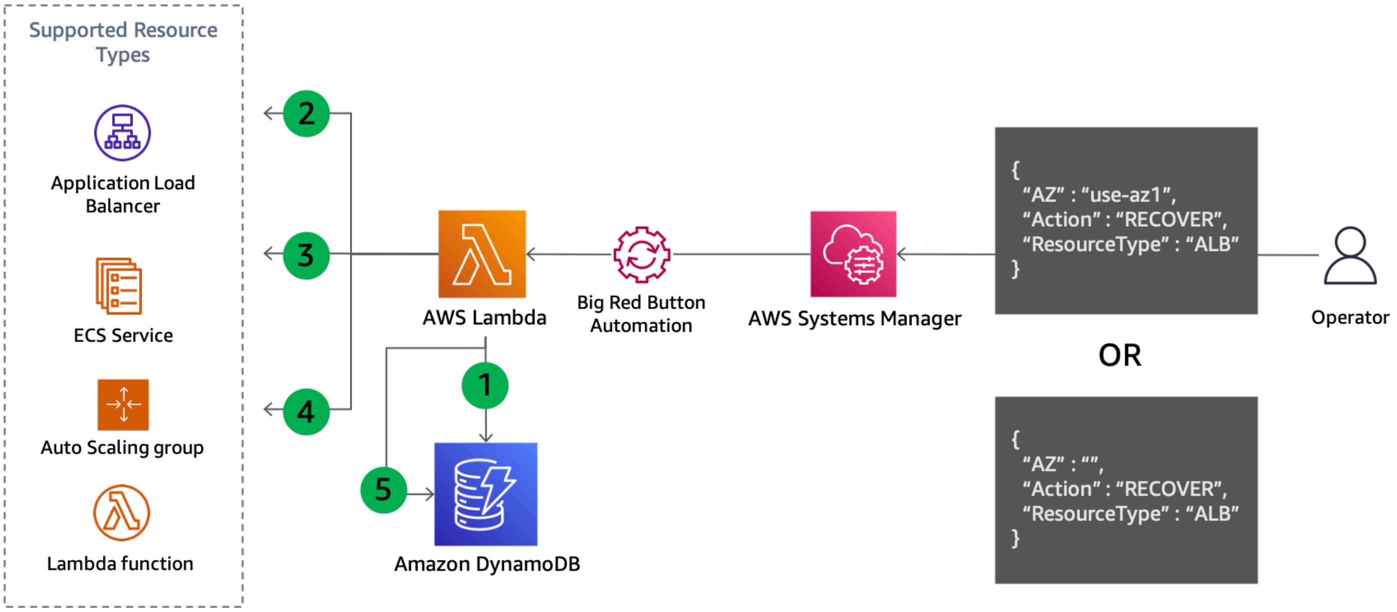


```
{
  "AZ" : "use-az1",
  "Action" : "EVACUATE",
  "ResourceType" : "ALB"
}
```

Control plane update to evacuate an Availability Zone

1. The script lists all of the resources of the specified type, such as Auto Scaling group, ECS service, or Lambda function, and retrieves their subnets from the resource information. The supported resources depend on what the script has been configured to support.
2. It determines which subnets should be removed by comparing each subnet's Availability Zone name to its mapped Availability Zone ID that was provided as an input parameter.
3. The network configuration of the resource is updated to remove the identified subnets.
4. The details of the update are recorded in a DynamoDB table. The Availability Zone ID is stored as the partition key and the resource ARN or name is stored as the sort key. The subnets that were removed are stored as a string array. Finally, the resource type is also stored and used as a hash key for a Global Secondary Index (GSI).

Because step four records the updates that were made, this approach also lends itself to being easily reversible when you're ready to recover, as shown in the following figure.



```
{
  "AZ" : "use-az1",
  "Action" : "RECOVER",
  "ResourceType" : "ALB"
}
```

```
{
  "AZ" : "",
  "Action" : "RECOVER",
  "ResourceType" : "ALB"
}
```

Control plane update to recover from Availability Zone evacuation

Recovery steps:

1. Query the GSI to get the subnets removed for each resource of the specified type in the specified Availability Zone (or all Availability Zones if one isn't specified).
2. Describe each resource found in the DynamoDB query to get its current network configuration.
3. Combine the subnets from the current network configuration with those retrieved from the DynamoDB query.
4. Update the network configuration of the resource with the new subnet set.
5. Remove the record from the DynamoDB table after the update completes successfully.

This generalized pattern both prevents routing work to the impacted Availability Zone and prevents new capacity from being deployed there. The following are examples of how this is accomplished for different services.

- **Lambda** — Update the function's VPC configuration to remove the subnets in the specified Availability Zone.
- **Auto Scaling Group** — Remove the subnets from the ASG configuration which will replace that capacity in the remaining Availability Zones.
- **Amazon ECS** — Update the ECS service VPC configuration to remove the subnets.

- **Amazon EKS** — Apply taints to nodes in the impacted Availability Zone to evict existing pods and prevent additional pods from being scheduled there.

Each service will react differently to the configuration update. For example, Amazon ECS will follow the service's deployment configuration after an update and trigger a rolling deployment or blue/green deployment of new tasks.

These updates may shift work to the healthy Availability Zones too quickly for some workloads. While being configured to be statically stable to the failure (having enough capacity pre-provisioned in the remaining Availability Zones to handle the impacted Availability Zone's work), you may also want to gradually phase out capacity from the impacted Availability Zone.

If you plan to update the network configuration of your Auto Scaling group that is a target group for a load balancer with cross-zone load balancing disabled, follow this guidance.

Auto Scaling reacts to this change using its Availability Zone rebalancing logic. It will launch instances in the other Availability Zones to meet your desired capacity and terminate instances in the Availability Zone you removed. However, the load balancer will continue to split traffic evenly across each Availability Zone, including the one you removed from the ASG, while the instances are being terminated. This could lead to a brown out of the remaining capacity in that Availability Zone until all instances are successfully terminated there. This is the same problem described in *Availability Zone independence* concerning Availability Zone imbalance when cross-zone load balancing is disabled. To prevent this from occurring, you can either:

- Always perform your Availability Zone evacuation first so traffic is only being split among the remaining Availability Zones
- Specify a minimum healthy target count with DNS failover to match your required minimum target count for that Availability Zone.

This will help ensure traffic is not sent to the Availability Zone you removed after instances start being terminated.

Summary

The following table summarizes the pros and cons of the evacuation patterns described.

Table 5: Evacuation pattern pros and cons

Approach	Pros	Cons
Data plane-controlled evacuation	Relies only on data plane actions Quickly prevents work from being done in the impacted Availability Zone Flexible approach to a centralized view of Availabil ity Zone health	Does not prevent capacity from being deployed in an impacted Availability Zone Not all workload types can use this approach easily
Control plane-controlled evacuation	Prevents new capacity from being deployed in the impacted Availability Zone Removes existing capacity from the impacted Availabil ity Zone	Relies on each service's control plane Requires code to be written for each service Has to be completed service by service Needs to be careful not to overwhelm capacity during the update

You will likely use both approaches together as part of an Availability Zone evacuation plan. Start with the data plane-controlled evacuation actions that are more likely to succeed to quickly stop processing work in the impacted Availability Zone. Then, once the initial impact is mitigated, follow-up with the control plane-controlled evacuation actions, if you deem it necessary.

Conclusion

This paper provided an overview of gray failures, how they manifest, and outlined why you need to build observability and evacuation tooling to mitigate those types of events when they occur. In the next section, you reviewed multi-AZ observability and three approaches you can implement to detect single Availability Zone impact. In the last section, this paper presented two general approaches for performing Availability Zone evacuation. The first approach uses data plane actions to prevent work from being routed to the impacted Availability Zone while the second approach uses control plane actions to prevent capacity from being provisioned in the impacted Availability Zone. Together, these two approaches achieve the two outcomes that Availability Zone evacuation intends.

The recovery patterns described in this paper will likely be part of a larger monitoring and fault recovery solution. This approach to dealing with single-Availability Zone gray failures requires engineering work to build the instrumentation necessary to detect them as well as the tooling to respond to them. However, for many workloads, this approach can be a simpler and less costly alternative to building multi-Region architectures. Additionally, it can help achieve smaller RPOs and RTOs (which increases the workload’s availability) when compared to multi-Region DR.

Appendix A – Getting the Availability Zone ID

If you are using the AWS .NET SDK (as well as some others like JavaScript) or running your system on an EC2 instance (including Amazon ECS and Amazon EKS), you can get the Availability Zone ID directly.

- **AWS .NET SDK**

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

- **EC2 Instance Metadata Service**

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

  On other platforms, such as Lambda and Fargate, you will need to retrieve the Availability Zone name and then find the mapping to the Availability Zone ID. With the Availability Zone name you can find the Availability Zone ID like this:

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

  The following examples to find the Availability Zone name to be used in the example above are written in bash using the AWS CLI and the package jq. They will need to be converted to the programming language used for your workload.

  - **Amazon ECS** - If the Instance Metadata Service (IMDS) is blocked by the host, you can use the container metadata file instead.

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

  - **Fargate** (platform version 1.4 or later)

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

- **Lambda –** The Availability Zone is not exposed directly to the function. To find it, you need to complete several steps. To do this, you will need to build a private API Gateway REST endpoint that returns the IP address of the requestor. This will identify the private IP assigned to the elastic network interface being used by the function.
  - Call the Lambda `GetFunction` API to find the VPC ID of the function.
  - Call the API Gateway service to get the function’s IP.
  - Using the IP and VPC ID, find the associated network interface and extract the Availability Zone.

```
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’)
```

Appendix B – Example chi-squared calculation

The following is an example of collecting error metrics and performing a chi-squared test on the data. The code is not production ready and does not perform necessary error handling, but does provide a proof of concept on how the logic works. You should update this example to fit your needs.

First, a Lambda function is invoked each minute by an Amazon EventBridge scheduled event. The content of the event is configured with the following data:

```
{
  "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" ]
}
```

The data is used to specify the common data needed to retrieve the appropriate CloudWatch metrics (like namespace, metric name, and dimensions) and then publish the chi-squared results for each Availability Zone. The code in the Lambda function looks like the following using Python 3.9. At a high level, it collects the specified CloudWatch metrics for the previous minute, runs the chi-squared test on that data, and then publishes CloudWatch metrics about the result of the test for each Availability Zone specified.

```
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)
```

You can then create an alarm per AZ. The following example is for `use1-az2` and alarms for three, one-minute data points in a row that have a maximum value equal to 1 (1 is the metric being published when the chi-squared test determines statistically significant skew in the error rate).

```
{
    "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"
    }
}
```

You can also create an m-of-n alarm and combine these two alarms together with a composite alarm. You would also need to create the same alarms for each Controller/Action combination or microservice you have in each Availability Zone. Finally, you can add the chi-squared composite alarm to the Availability Zone-specific alarm for each Controller/Action combination as shown in Failure detection using outlier detection.

Contributors

Contributors to this document include:

- Michael Haken, Principal Solutions Architect, Amazon Web Services

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Whitepaper updated	Updated with additional observability guidance and to use the new zonal shift feature.	July 11, 2023
Initial publication	Whitepaper first published.	March 2, 2022

ⓘ **Note**

To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser you are using.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.