Static stability using Availability Zones

Becky Weiss, Mike Furr

At Amazon, the services we build must meet extremely high availability targets. This means that we need to think carefully about the dependencies that our systems take. We design our systems to stay resilient even when those dependencies are impaired. In this article, we'll define a pattern that we use called *static stability* to achieve this level of resilience. We'll show you how we apply this concept to Availability Zones, a key infrastructure building block in AWS and therefore a bedrock dependency on which all of our services are built.

In a statically stable design, the overall system keeps working even when a dependency becomes impaired. Perhaps the system doesn't see any updated information (such as new things, deleted things, or modified things) that its dependency was supposed to have delivered. However, everything it was doing before the dependency became impaired continues to work despite the impaired dependency. We'll describe how we built Amazon Elastic Compute Cloud (EC2) to be statically stable. Then we'll provide two statically stable example architectures we've found useful for building highly available regional systems on top of Availability Zones.

Finally, we'll go deeper into some of the design philosophy behind Amazon EC2 including how it's architected to provide Availability Zone independence at the software level. In addition, we'll discuss some of tradeoffs that come with building a service with this choice of architecture.

The role of Availability Zones

Availability Zones are logically isolated sections of an AWS Region: Each AWS Region has multiple Availability Zones that are designed to operate independently. Availability Zones are physically separated by a meaningful distance to protect from correlated impact from potential issues such as lightning strikes, tornadoes, and earthquakes. They don't share power or other infrastructure, but they are connected to each other with fast, encrypted private fiber-optic networking to enable applications to quickly fail over without interruption. In other words, Availability Zones provide a layer of abstraction over our infrastructure isolation. Services that require an Availability Zone allow the caller to tell AWS where to provision the infrastructure physically within the Region so that they can benefit from this independence. At Amazon, we have built regional AWS services that take advantage of this zonal independence to achieve their own high availability targets. Services such as Amazon DynamoDB, Amazon Simple Queue Service (SQS), and Amazon Simple Storage Service (S3) are examples of such regional services.

When interacting with an AWS service that provisions cloud infrastructure inside of an Amazon Virtual Private Cloud (VPC), many of these services require the caller to specify not only a Region but also an Availability Zone. The Availability Zone is often specified implicitly in a required subnet argument, for example when launching an EC2 instance, provisioning an Amazon Relational Database Service (RDS) database, or creating an Amazon ElastiCache cluster. Although it's common to have multiple subnets in an Availability Zone, a single subnet lives entirely within a single Availability Zone, and so by providing a subnet argument, the caller is also implicitly providing an Availability Zone to use.

Static stability

When building systems on top of Availability Zones, one lesson we have learned is to be ready for impairments before they happen. A less effective approach might be to deploy to multiple Availability Zones with the expectation that, should there be an impairment within one Availability

Zone, the service will scale up (perhaps using AWS Auto Scaling) in other Availability Zones and be restored to full health. This approach is less effective because it relies on reacting to impairments as they happen, rather than being prepared for those impairments before they happen. In other words, it lacks static stability. In contrast, a more effective, statically stable service would overprovision its infrastructure to the point where it would continue operating correctly without having to launch any new EC2 instances, even if an Availability Zone were to become impaired.

To better illustrate the property of static stability, let's look at Amazon EC2, which is itself designed according to those principles.

The Amazon EC2 service consists of a control plane and a data plane. “Control plane” and “data plane” are terms of art from networking, but we use them all over the place within AWS. A *control plane* is 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. A *data plane*, in contrast, is the daily business of those resources, that is, what it takes for them to function.

In Amazon EC2, the control plane is everything that happens when EC2 launches a new instance. The logic of the control plane pulls together everything needed for a new EC2 instance by performing numerous tasks. The following are a few examples:

- It finds a physical server for the compute while respecting placement group and VPC tenancy requirements.
- It allocates a network interface out of the VPC subnet.
- It prepares an Amazon Elastic Block Store (EBS) volume.
- It generates AWS Identity and Access Management (IAM) role credentials.
- It installs Security Group rules.
- It stores the results in the data stores of the various downstream services.
- It propagates the needed configurations to the server in the VPC and the network edge as appropriate.

In contrast, the Amazon EC2 data plane keeps existing EC2 instances humming along as expected, performing tasks such as these:

- It routes packets according to the VPC's route tables.
- It reads and writes from Amazon EBS volumes.
- And so on.

As is usually the case with data planes and control planes, the Amazon EC2 data plane is far simpler than its control plane. As a result of its relative simplicity, the Amazon EC2 data plane's design targets a higher availability than that of the Amazon EC2 control plane.

Importantly, the Amazon EC2 data plane has been carefully designed to be statically stable in the face of control plane availability events (such as impairments in the ability to launch EC2 instances). For example, to avoid disruptions in network connectivity, the Amazon EC2 data plane is designed so that the physical machine on which an EC2 instance runs has local access to all of the information it needs to route packets to points inside and outside of its VPC. An impairment of the Amazon EC2 control plane means that during the event the physical server might not see updates like a new EC2 instance added to a VPC, or a new Security Group rule. However, the traffic it had been able to send and receive before the event will continue to work.

The concepts of control planes, data planes, and static stability are broadly applicable, even beyond Amazon EC2. Being able to decompose a system into its control plane and data plane can be a helpful conceptual tool for designing highly available services, for a number of reasons:

- It's typical for the availability of the data plane to be even more critical to the success of the customers of a service than the control plane. For example, the continued availability and correct functioning of an EC2 instance, after it is running, is even more important to most AWS customers than the ability to launch new EC2 instances.
- It's typical for the data plane to operate at a higher volume (often by orders of magnitude) than its control plane. Thus, it's better to keep them separate so that each can be scaled according to its own relevant scaling dimensions.
- We've found over the years that a system's control plane tends to have more moving parts than its data plane, so it's statistically more likely to become impaired for that reason alone.

Putting those considerations all together, our best practice is to separate systems along the control and data plane boundary.

To achieve this separation in practice, we apply principles of static stability. A data plane typically depends on data that arrives from the control plane. However, to achieve a higher availability target, the data plane maintains its existing state and continues working even in the face of a control plane impairment. The data plane might not get updates during the period of impairment, but everything that had been working before continues to work.

Earlier we noted that a scheme that requires the replacement of an EC2 instance in response to an Availability Zone impairment is a less effective approach. It's not because we won't be able to launch the new EC2 instance. It's because in response to an impairment the system has to take an immediate dependency for the recovery path on the Amazon EC2 control plane, plus all of the application-specific systems that are necessary for a new instance to start performing useful work. Depending on the application, these dependencies could include steps such as downloading runtime configuration, registering the instance with discovery services, acquiring credentials, etc. The control plane systems are necessarily more complex than those in the data plane, and they have a greater chance of not behaving correctly when the overall system is impaired.

Static stability patterns

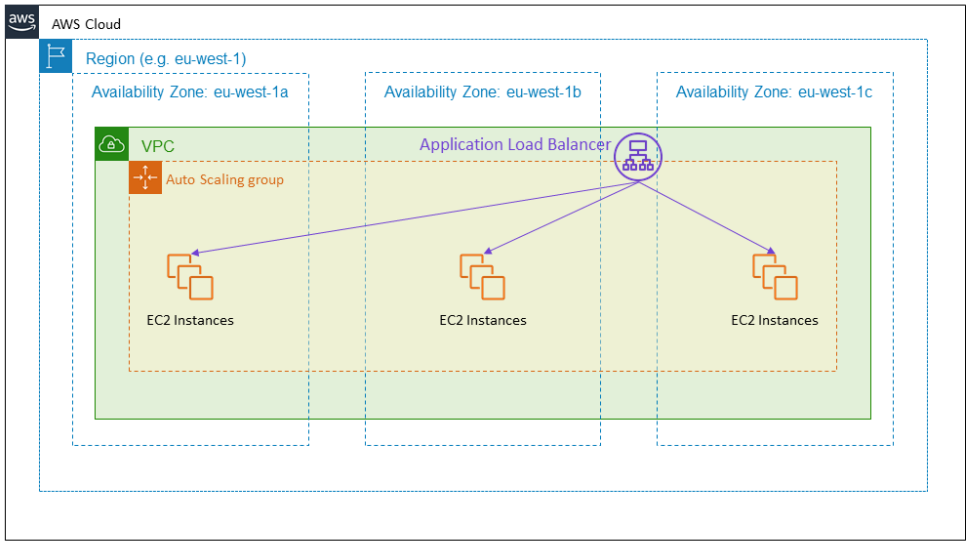
In this section, we'll introduce two high-level patterns that we use in AWS to design systems for high availability by leveraging static stability. Each is applicable to its own set of situations, but both take advantage of the Availability Zone abstraction.

Active-active on Availability Zones example: A load-balanced service

Several AWS services are internally composed of a horizontally scalable, stateless fleet of EC2 instances or Amazon Elastic Container Service (ECS) containers. We run these services in an Auto Scaling group across three or more Availability Zones. Additionally, these services overprovision capacity so that, even if an entire Availability Zone were impaired, the servers in the remaining Availability Zones could carry the load. For example, when we use three Availability Zones, we overprovision by 50 percent. Put another way, we overprovision such that each Availability Zone is

operating at only 66 percent of the level for which we have load-tested it.

The most common example is a load balanced HTTPS service. The following diagram shows a public-facing Application Load Balancer providing an HTTPS service. The target of the load balancer is an Auto Scaling group that spans the three Availability Zones in the eu-west-1 Region. This is an example of active-active high availability using Availability Zones.



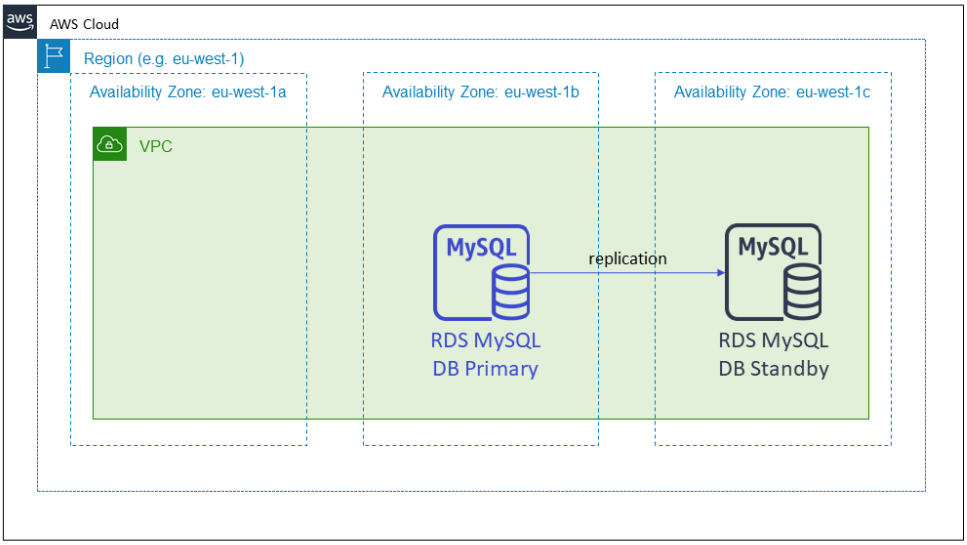
In the event of an Availability Zone impairment, the architecture shown in the preceding diagram requires no action. The EC2 instances in the impaired Availability Zone will start failing health checks, and the Application Load Balancer will shift traffic away from them. In fact, the Elastic Load Balancing service is designed according to this principle. It has provisioned enough load-balancing capacity to withstand an Availability Zone impairment without needing to scale up.

We also use this pattern even when there is no load balancer or HTTPS service. For example, a fleet of EC2 instances that processes messages from an Amazon Simple Queue Service (SQS) queue can follow this pattern too. The instances are deployed in an Auto Scaling group across multiple Availability Zones, appropriately overprovisioned. In the event of an impaired Availability Zone, the service does nothing. The impaired instances stop doing their work, and others pick up the slack.

Active-standby on Availability Zones example: A relational database

Some of the services we build are stateful and require a single primary or leader node to coordinate the work. An example of this is a service that uses a relational database, such as Amazon RDS with a MySQL or Postgres database engine. A typical high-availability setup for this kind of relational database has a primary instance, which is the one to which all writes must go, and a standby candidate. We also might have additional read replicas, which are not shown in the following diagram. When we work with stateful infrastructure like this, there will be a warm standby node in a different Availability Zone from that of the primary node.

The following diagram shows an Amazon RDS database. When we provision a database with Amazon RDS, it requires a subnet group. A *subnet group* is a set of subnets spanning multiple Availability Zones into which the database instances will be provisioned. Amazon RDS puts the standby candidate in a different Availability Zone from the primary node. This is an example of active-standby high availability using Availability Zones.



As was the case with the stateless, active-active example, when the Availability Zone with the primary node becomes impaired, the stateful service does nothing with the infrastructure. For services that use Amazon RDS, RDS will manage the failover and repoint the DNS name to the new primary in the working Availability Zone. This pattern also applies to other active-standby setups, even if they do not use a relational database. In particular, we apply this to systems with a cluster architecture that has a leader node. We deploy these clusters across Availability Zones and elect the new leader node from a standby candidate instead of launching a replacement "just in time."

What these two patterns have in common is that both of them had already provisioned the capacity they'd need in the event of an Availability Zone impairment well in advance of any actual impairment. In neither of these cases is the service taking any deliberate control plane dependencies, such as provisioning new infrastructure or making modifications, in response to an Availability Zone impairment.

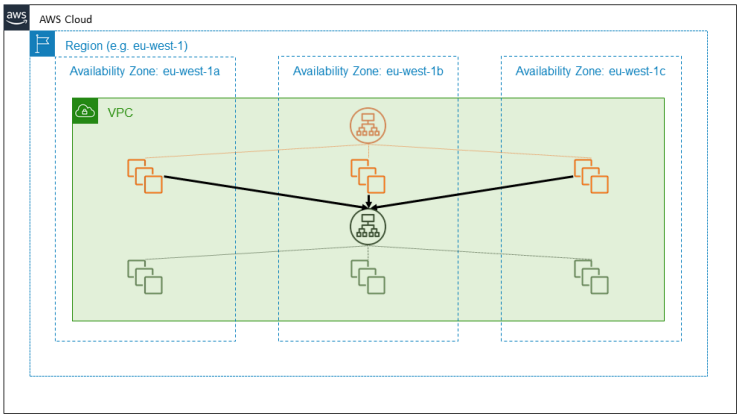
Under the hood: Static stability inside of Amazon EC2

This final section of the article will go one level deeper into resilient Availability Zone architectures, covering some of the ways in which we follow the Availability Zone independence principle in Amazon EC2. Understanding some of these concepts is helpful when we build a service that not only needs to be highly available itself, but also needs to provide infrastructure on which others can be highly available. Amazon EC2, as a provider of low-level AWS infrastructure, is the infrastructure that applications can use to be highly available. There are times when other systems might wish to adopt that strategy as well.

We follow the Availability Zone independence principle in Amazon EC2 in our deployment practices. In Amazon EC2, software is deployed to the physical servers hosting EC2 instances, edge devices, DNS resolvers, control plane components in the EC2 instance launch path, and many other components upon which EC2 instances depend. These deployments follow a zonal deployment calendar. This means that two Availability Zones in the same Region will receive a given deployment on different days. Across AWS, we use a phased rollout of deployments. For example, we follow the best practice (regardless of the type of service to which we deploy) of first deploying a one-box, and then 1/N of servers, etc. However, in the specific case of services like those in Amazon EC2, our deployments go one step further and are deliberately aligned to the Availability Zone boundary. That way, a problem with a deployment affects one Availability Zone and is rolled back and fixed. It doesn't affect other Availability Zones, which continue functioning as normal.

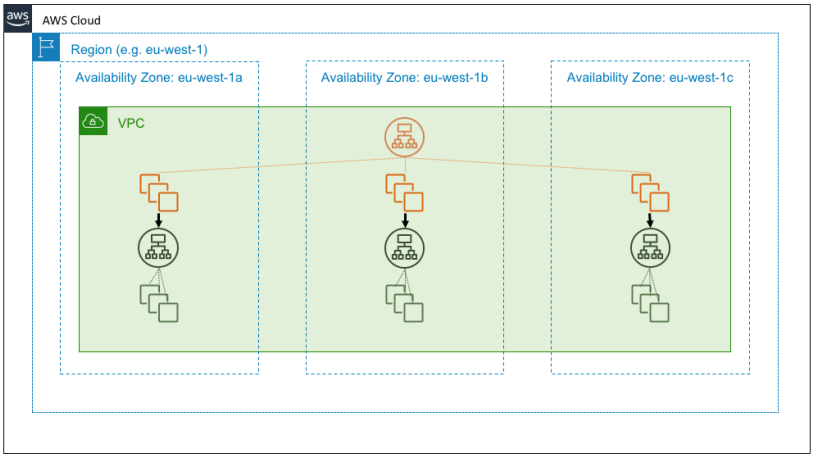
Another way we use the principle of independent Availability Zones when we build in Amazon EC2 is to design all packet flows to stay within the Availability Zone rather than crossing boundaries. This second point—that network traffic is kept local to the Availability Zone—is worth exploring in more detail. It's an interesting illustration of how we think differently when building a regional, highly available system that is a consumer of independent Availability Zones (that is, it uses guarantees of Availability Zone independence as a foundation for building a high-availability service), as opposed to when we provide Availability Zone independent infrastructure to others that will allow them to build for high availability.

The following diagram illustrates a highly available external service, shown in orange, that depends on another, internal service, shown in green. A straightforward design treats both of these services as consumers of independent EC2 Availability Zones. Each of the orange and green services is fronted by an Application Load Balancer, and each service has a well-provisioned fleet of backend hosts spread across three Availability Zones. One highly available regional service calls another highly available regional service. This is a simple design, and for many of the services we've built, it's a good design.



Suppose, however, that the green service is a foundational service. That is, suppose it is intended not only to be highly available but also, itself, to serve as a building block for providing Availability Zone independence. In that case, we might instead design it as three instances of a zone-local service, on

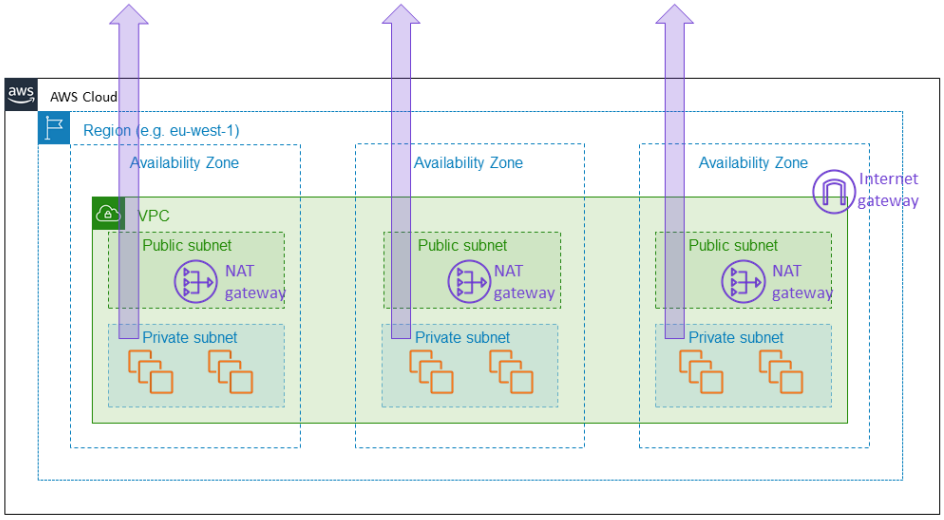
which we follow Availability Zone-aware deployment practices. The following diagram illustrates the design in which a highly available regional service calls a highly available zonal service.



The reasons why we design our building-block services to be Availability Zone independent come down to simple arithmetic. Let's say an Availability Zone is impaired. For black and white failures, the Application Load Balancer will automatically fail away from the affected nodes. However, not all failures are so obvious. There can be gray failures, such as bugs in the software, which the load balancer won't be able to see in its health check and cleanly handle.

In the earlier example, where one highly available regional service calls another highly available regional service, if a request is sent through the system, then with some simplifying assumptions, the chance of the request avoiding the impaired Availability Zone is 2/3 * 2/3 = 4/9. That is, the request has worse than even odds of steering clear of the event. In contrast, if we built the green service to be a zonal service as in the current example, then the hosts in the orange service can call the green endpoint in the same Availability Zone. With this architecture, the chances of avoiding the impaired Availability Zone are 2/3. If N services are a part of this call path, then these numbers generalize to (2/3)^N for N regional services versus remaining constant at 2/3 for N zonal services.

It is for this reason that we built Amazon EC2 NAT gateway as a zonal service. NAT gateway is an Amazon EC2 feature that allows for outbound internet traffic from a private subnet and appears not as a regional, VPC-wide gateway but as a zonal resource, that customers instantiate separately per Availability Zone as shown in the following diagram. The NAT gateway sits in the path of internet connectivity for the VPC, and it is therefore part of the data plane of any EC2 instance within that VPC. If there is a connectivity impairment in one Availability Zone, we want to keep that impairment inside that Availability Zone, rather than spreading it to other zones. In the end, we want a customer who built an architecture similar to that mentioned earlier in this article (that is, by provisioning a fleet across three Availability Zones with enough capacity in any two to carry the full load) to know that the other Availability Zones will be completely unaffected by anything going on in the impaired Availability Zone. The only way for us to do this is to ensure that all foundational components—like the NAT gateway—really do stay within the Availability Zone.



This choice comes with the cost of additional complexity. For us in Amazon EC2, the additional complexity comes in the form of managing zonal, rather than regional, service environments. For customers of NAT gateway, the additional complexity comes in the form of having multiple NAT gateways and route tables, for use in the different Availability Zones of the VPC. The additional complexity is appropriate because NAT gateway is itself a foundational service, part of the Amazon EC2 data plane that is supposed to provide zonal availability guarantees.

There's one more consideration we make when building services that are Availability Zone independent, and that is data durability. Although each of the zonal architectures described earlier shows the entire stack contained within a single Availability Zone, we replicate any hard state across multiple Availability Zones for disaster recovery purposes. For example, we typically store periodic database backups in Amazon S3 and maintain read replicas of our data stores across Availability Zone boundaries. These replicas aren't necessary for the primary Availability Zone to function. Instead, they ensure that we store customer- or business-critical data in multiple locations.

When designing a service-oriented architecture that will run in AWS, we have learned to use one of these two patterns, or a combination of both:

- The simpler pattern: regional-calls-regional. This is often the best choice for external-facing services, and appropriate for most internal services as well. For example, when building higher-level application services in AWS, such as Amazon API Gateway and AWS serverless technologies, we use this pattern to provide high availability even in the face of Availability Zone impairments.
- The more complex pattern: regional-calls-zonal or zonal-calls-zonal. When designing internal, and in some cases external, data plane components within Amazon EC2 (for example, network appliances or other infrastructure that sits directly in the critical data path) we follow the pattern of Availability Zone independence and use instances that are siloed in Availability Zones, so that network traffic remains in its same Availability Zone. This pattern not only

helps keep impairments isolated to an Availability Zone but also has favorable network traffic cost characteristics in AWS.

Conclusion

In this article, we've discussed some simple strategies that we've used at AWS for successfully taking dependencies on Availability Zones. We've learned that the key to static stability is to anticipate impairments before they happen. Whether a system runs on an active-active horizontally scalable fleet, or whether it is a stateful, active-standby pattern, we can use Availability Zones to target high levels of availability. We deploy our systems so that all capacity that will be needed in the event of an impairment is already fully provisioned and ready to go. Finally, we took a deeper look into how Amazon EC2 itself uses static stability concepts to keep Availability Zones independent of one another.