Application Design for Availability

five most common practices we apply to improve availability are following: • Fault Isolation Zones • Redundant components • Micro-service architecture • Recovery Oriented Computing • Distributed systems best practices

**Fault Isolation Zones**

make use of multiple independent components in parallel (Example : AWS Availability Zones.)

in the case of AWS Regions, autonomously

AWS has multiple constructs that provide different levels of independent, redundant components

* AWS partitions resources and requests via some dimension, such as a resource ID. These partitions (which we refer to as “cells” but others may call “shards” or “stripes”) are designed to be independent and further contain faults to within a single cell.
* Availability zone
  + Each Availability Zone has independent physical infrastructure: dedicated connections to utility power, standalone backup power sources
  + geographically separated
  + Same region - enables synchronous data replication – no impact on application latency

allows customers to use Availability Zones in an active/active or active/standby configuration

* AWS Region. Regions are designed to be autonomous, with dedicated copies of services deployed in each Region.
  + AWS allows cross-Region services – example – S3 replication, image/snapshots copy.
  + Global services – IAM, Route53, CloudFront

**Redundant Components**

avoidance of single points of failure in underlying physical infrastructure

using multiple AS and resilient to failure of a single zone

systems are built to be resilient to failure of a single compute node, single storage volume, or single instance of a database.

**Micro-Service Architecture**

AWS services uses micro-services

micro-services are smaller and simpler. They allow you to differentiate the availability required of different services, and thereby focus investments more specifically to the micro-services that have the greatest availability needs

Example : Amazon product details page – lot of microservies are invoked – impt services like price/product details – customer should be able to buy. This functionality is critical – focus investment on this – HA required for this.

Pros v/s cons

**Recovery-Oriented Computing**

* Improve recovery

work to minimize the disruption time when failures do occur.

Since impact duration is a primary input to calculating availability, reducing recovery time has a direct impact on improving availability

ROC identifies the characteristics in systems that enhance recovery:

* isolation and redundancy (Fault Isolation zone and redundant components)
* modular design (Micro-service architecture)
* monitoring for health,
* automated recovery
* ability to restart
* ability to roll back changes,
* monitoring,
* diagnostics

Need to have a mechanism to detect failures and apply one of the well-tested recovery path

* In systems that apply a recovery-oriented approach, many different categories of failures are mapped to the same recovery strategy

Example : Network timeout & dependency failure where dependency returns an error

Solution – Retry

* An instance may fail due to hardware failure, operating system bug, memory leak, or other causes. Rather than building custom remediation for each, treat any as an instance failure, terminate the instance, and allow Auto Scaling to replace the instance

Solution : Add another instance using Auto scaling

* A pattern to avoid is developing recovery paths that are rarely executed

Failover from primary to secondary datastore/db

establish recovery patterns and regularly test them. If you have a complex or critical recovery path, you still need to regularly execute that failure in production to convince yourself that the recovery path works.

* design services to minimize the time to recover from failures and impact on data **(Ex: All databases, S3,SQS)**

**Distributed Systems Best Practices**

many systems built today are distributed systems. They rely on communications networks to interconnect components

these systems can have high latency or loss

Individual services may see spikes of requests that temporarily overwhelm their ability to respond

There are a number of best practices that can be applied to allow these services to continue to operate normally in the presence of these “normal” issues.

Throttling:

* defensive pattern to respond to an unexpected increase in
* Some requests will be honored, but the rejected requests will return a message indicating they have been throttled, with the expectation they will try again at a slower rate
* Your services should be designed to a known capacity of requests that each node or cell can process – using load testing..
* User will retry –
* **Example : AWS API Gateway**

Retry with exponential fallback

* The pattern is to pause and then retry at a later time. If it fails again, pause longer and retry. This increase in pause time is often called “backing off.” After a configured number of attempts or elapsed time, it will quit retrying and return failure

Fail fast:

Simply return an error as soon as possible. This will allow releasing of resources associated with requests and can often allow a service to recover if it is running out of resources. It’s preferable to fail fast rather than allowing requests to be queued.

Use of idempotency tokens

**hard to guarantee an action is performed exactly once**

Callers issue API requests with an idempotency token; the same token is used whenever the request is repeated (for example, due to a timeout and retry.) When receiving a request that has already been processed, an idempotent API uses the token to determine the work has already been completed, and then returns a response identical to the response that’s returned when the work is completed for the first time.

Constant work:

Design system to constantly perform work at max desired capacity

Circuit breaker:

Service  calling a dependent service. In failure is returned or high latency,

* Open circuit
* Use the cache response, do not make dependent service call
* Periodically check the dependent service
* If OK, close the circuit

Bi-modal behavior and static stability

Distributed systems can be impacted by negative feedback loops that are triggered by one failure

we prefer building systems that are statically stable and operate in only one mode. They maintain enough internal state to continue operating as they were before the failure without adding additional load to the system

Example : Another example of this type of system is one that uses Amazon EC2 for instance capacity. Systems often assume that if an instance or Availability Zone fails, they will respond by simply launching new instances. However, this approach means that during failure, the system will be doing much different work from usual. Instead, we recommend using Elastic Load Balancing or Amazon Route53 health checks to shift load away from failed instances, and use Auto Scaling to asynchronously replace them.

**Operational Considerations for Availability**

Testing is an important part of the delivery pipeline. Aside from common unit tests and functional tests that be performed at component levels, it is important to perform sustained load testing, performance testing, and fault injection testing.

In addition, your monitoring service must be able to add or remove monitoring of capabilities that are added or deprecated.

**Automate Deployments to Eliminate Impact**

use of automation wherever practical in operations, including testing and deploying changes, adding or removing capacity, and migrating data

These are deployment patterns that minimize risk: • Canary deployment • Blue-Green deployment • Feature toggles • Failure isolation zone deployments

Canary deployment

practice of directing a small number of your customers to the new version and scrutinizing deeply any behavior changes or errors that are generated

remove traffic from the canary if you have critical problems and send the users to the previous version

* Using **AWS Code Deploy**

Blue-Green Deployments

similar to the canary deployment except that a full fleet of the application is deployed in parallel

2 deployments running in parallel

Feature toggles

* configuration options to turn on/off a new feature..

**Testing**

Testing to ensure that you can meet your availability goals is the only way you can have confidence that you will meet those goals.

Our experience is that canary testing that can run constantly and simulate customer behavior is among the most important testing processes.

You should unit test, load test, performance test, and simulate your failure modes while under these tests. Don’t forget to test for external dependency unavailability, and deployment failures

Other modes of degradation may cause reduced functionality and slow responses,

ability to inject random failures into your system, including component failures, networking effects such as latency and dropped messages, and DNS failures such as being unable to resolve a name or not being able to establish connections to dependent services

Netflix – Semian army

**Monitoring and Alarming**

Monitoring at AWS consists of five distinct phases: 1. Generation 2. Aggregation 3. Real-time processing and alarming 4. Storage 5. Analytics

Generation

determine which services and/or applications require monitoring, define important metrics and how to extract them from log entries if necessary, and finally create thresholds and corresponding alarm event

Aggregation

Amazon CloudWatch and Amazon S3 serve as the primary aggregation and storage layers

Real-Time Processing and Alarming

Alerts can trigger *Auto Scaling events*, so that clusters react to changes in demand

Alerts can be sent to  SNS topics, SQS, Lambda

There are a number of tools provided by partners and third parties that allow for aggregation, processing, storage, and analytics. Some of these tools are New Relic, Splunk, Loggly, Logstash, CloudHealth, and Nagio

Amazon S3 supports lifecycle management at the S3 bucket level. This lifecycle management can be applied differently to different paths in the bucket. Toward the end of the lifecycle you can transition data to Amazon Glacier for long-term storage, and then expiration, after the end of the retention period is reached.

Key AWS services

**CloudWatch**

**AWS X-Ray** - can be integrated with your applications to provide visibility into the distributed interaction of requests with your applications

Amazon S3: Acts as the storage layer, and allows for lifecycle policies and data management.

Amazon EMR: Use this service to gain further insight into log and metric data

Storage and Analytics

*S3 – storage*

*EMR – analytics*

**Operational Readiness Reviews**

important exercise to confirm applications are ready for production operations

A formal ORR is conducted prior to initial production deployment

An ORRs for one application should incorporate lessons learned and best practices from other applications

**Auditing**

For root cause analysis

Amazon CloudWatch Logs: You can store your logs in this service and inspect their contents.

AWS Config: You can see what AWS infrastructure were used at points in time.

AWS CloudTrail: You can see which AWS APIs were invoked at what time and by what principal

Example Implementations for Availability Goals

Single Region Scenarios

Multi-Region Scenarios