IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 25 June 2023, accepted 31 July 2023, date of publication 9 August 2023, date of current version 16 August 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3303430

RESEARCH ARTICLE

Evaluate Solutions for Achieving High Availability or Near Zero Downtime for Cloud Native Enterprise Applications

ANTRA MALHOTRA[1], (Member, IEEE), AMR ELSAYED[2], RANDOLPH TORRES[3], AND SRINIVAS VENKATRAMAN[4]

[1]System Engineering, Network Systems—Shared Platforms and Product, Verizon, Temple Terrace, FL 33637, USA
[2]System Architecture, Network Systems—Shared Platforms and Product, Verizon, Temple Terrace, FL 33637, USA
[3]Technical Strategy, Network Systems—Shared Platforms and Product, Verizon, Temple Terrace, FL 33637, USA
[4]System Engineering, Network Systems—Shared Platforms and Product, Verizon, Basking Ridge, NJ 07920, USA

Corresponding author: Antra Malhotra (antra.malhotra@verizon.com)

The study related to high availability for cloud native enterprise applications was conducted as part of the Verizon initiative to build resilient, highly available and cloud native applications under the leadership of Sebastien Jobert (Executive Director at Verizon. Email: sebastien.jobert@verizon.com). This study is supported and funded by the Shared Platforms and Product Organization, part of Network Systems in Verizon.

ABSTRACT In today's digital world, businesses heavily rely on systems for every aspect of their operations and product life cycle. Hence, it is important to have a strong ecosystem of applications to achieve operational efficiency and positive customer experience. High availability of mission and business critical applications is a necessity as any downtime or poor performance can have a negative impact on the revenue and operations of the organization. Applications transitioning to cloud are adopting modular design and distributed system architecture. As a result, the system complexity and the number of failure points have increased. One of the promises of cloud platforms is high availability by building redundancy in the application architecture. However, enterprises opting for cloud often struggle to define the right framework for high availability. In addition, even after redundancy is built at the application layer, building a similar redundant and resilient architecture at the database layer is challenging. For near zero downtime experience, applications should be able to perform automated application and database failover with minimum manual intervention. This could be a valuable feature for applications that are expected to be available 24/7. In this paper we will define a cloud native template architecture that enterprise applications can incorporate to be highly available and evaluate techniques to perform automatic database failover for a near zero downtime experience. We will then incorporate the database failover technique as part of the recommended application architecture to review the impact during planned maintenance activities and outages.

INDEX TERMS Cloud computing, database failover, high availability, zero downtime.

I. INTRODUCTION

Asystem is considered highly available when it stays operational and continues to operate and provide service as per the acceptable threshold for a higher percentage of time. The percentage here is the amount of time the system is expected to be available and this also determines the permissible downtime due to outages and maintenance activities. System availability is a non-functional requirement and can be defined based on the service availability [1]. It can be calculated as [1] and [5]:

Service Availability = Service Uptime/ (Service Uptime addressxxxx Service Outage)

Service Uptime: Duration of time system is available

Service Outage: Duration of time system is unavailable

We can consider a system or service as highly available when it functions without any downtime for 99.999% of the time; this calculation of "five nines' is considered close. In other words, the permissible downtime to cover for any planned and unplanned outages is 5 minutes and 16 secs a year [1], [2], [3], [4]. In the Fig. 1 below, the table has a break-up of the Availability Level and Average Yearly Downtime [1], [2], [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Nikhil Padhi.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Availability Level	Average Yearly Downtime
99%	87 hours, 40 minutes
99.5%	43 hours, 50 minutes
99.9%	8 hours, 46 minutes
99.95%	4 hours, 23 minutes
99.995%	26 minutes, 18 seconds
99.999%	5 minutes, 16 seconds
99.9999%	31.6 seconds

FIGURE 1. **Table showing the availability levels and the corresponding average yearly downtime.**

A highly available application is resilient, redundant and it leverages the failover mechanisms to recover and keep the experience seamless for the users. Now, with more and more applications and services migrating to the cloud, high availability is a key requirement from the cloud providers. Here are a few examples from 2022 where cloud outages impacted multiple users and organizations across regions. In 2022, Google performed a routine maintenance event on a software defined networking component and that led to an outage of 3 hours and 22 mins at its US West 1B region in Oregon [7]. Similarly, on July 28 2022, AWS experienced a power loss in a single availability zone of the US East 2 region in Ohio and though the outage lasted for 20 minutes it knocked down the Third-party services for up to three hours [7]. This shows that high availability is still a challenge even when cloud providers promise to deliver scalable and redundant infrastructure. There is much literature and research work done to evaluate multiple real-life scenarios of cloud outages [6], [7], [8], [9].

For applications migrating to the cloud, there are many high availability architecture solutions and best practices available. However, there is no one solution that fits all. This paper reviews the different principles to build a highly available and fault tolerant application and then recommends template architecture for read intensive and write intensive database applications. After the template architecture is defined, one of the challenges to achieve zero downtime is the ability of the database to automatically perform a failover. Ideally, all business-critical applications must have database resiliency within and across regions. Applications must be able to switch automatically within a region and must be able to switch across regions with a click of a button without any other human intervention. This paper also evaluates the different strategies for performing automatic database failover and identifies the best way to perform the failover with the least amount of human intervention and error rate. The technique identified is then incorporated as part of the recommended architecture and failure scenarios associated with planned outages are executed to assess the downtime and application availability.

II. BACKGROUND AND HISTORY

In the era that predates internet, availability of systems was desirable but not necessarily an entitlement as computers were used only when needed. The websites were operational primarily during the working hours of its brick-and-mortar counterpart. The applications were created for a specific user base and were not product agnostic. However, with internet there is an increase in distributed computation and ability to meet the needs of the users across the globe. Today, applications cater to multiple functions and users in different time zones. Hence, the necessity for the applications to operate 24/7 becomes a key expectation.

There is research done in this area to define high availability, thresholds and different ways to calculate high availability [1], [2], [3], [4], [5], [8]. In addition, the research includes in- depth study on the major outages that impacted cloud providers in last several years [6], [7], [8]. So, as business it is important to be prepared to manage application and database failover in case of any outage or maintenance activity. For applications migrating to cloud having a universal template architecture and recommendations that can be readily used as part of their migration journey will decrease the learning curve and give opportunities for the applications to take advantage of the available blueprint rather than re-inventing the wheel from the scratch.

This paper focuses on building that template architecture, and uses the definitions and metrics defined in the various literature work as a baseline to achieve high availability of 99.999% [1], [2], [3], [4], [5], [8]. The applications are profiled as read intensive and write intensive database applications for the template architecture. There is related research work done that recommends High Availability Proxy (HAProxy) as the load balancing tool for managing the traffic to the application layer [21], [22]. This paper builds further on this research by leveraging High Availability Proxy (HAProxy) and PgBouncer for automated database failover for a cloud native application. In addition, this paper simulates the recommended architecture for read intensive and write intensive database applications and executes the failover scenarios in a test environment to record the downtime and assess the feasibility of achieving high availability in a cloud environment.

III. GUIDING PRINCIPLES FOR APPLICATION ARCHITECTURE

Enterprise, cloud-based applications are studied to define the template architecture for high availability and to evaluate database failover techniques [9], [10]. Two database applications are considered:

Application (1) Read intensive database application which is a centralized enterprise location primary data platform. It provides a one-stop service for standardization and validation for domestic and international addresses.

Application (2) Write intensive database application that receives transactions from the users and data feeds to update the site locations, equipment/antenna database and perform project management for field operations.

As an initial step, the guiding principles for near zero downtime experience are defined in the section below and

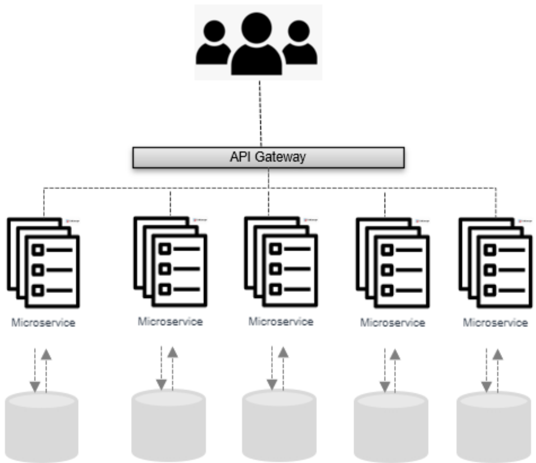


FIGURE 2. **Microservices based architecture showing modular design of the application.**

then keeping the Application (1) and Application (2) in context, the template architecture is recommended.

A. MICROSERVICES BASED ARCHITECTURE AND MODULAR DESIGN

Unlike monolithic design where all processes are coupled together to run as a single service, the microservice based architecture enables every function to run as an independent component. These components are micro-services that communicate by exposing APIs (Application Programming Interface) for other clients and microservices. Refer Fig. 2 below for the microservice based architecture showing a modular design of the application. The application layer provides the unified interface for these services. Each service can be managed, deployed, updated and scaled independently. These services do not share any code or implementation with other services. This modular design helps to break out the complex code into smaller chunks for easy maintainability. This supports fault isolation as the failures can be independently addressed or replaced. Multiple instances of the application can be deployed and using load balancing capabilities the traffic can be diverted to the available module, thereby increasing the redundancy and resilience of the system [9], [11], [12], [13], [14].

B. CLUSTERED ARCHITECTURE FOR APPLICATION

A clustered application enables load distribution and failover in case of any service disruption. Our objective is to utilize a cloud native application architecture as mentioned in Fig. 3 along with a Kubernetes (K8S) cluster infrastructure to balance traffic across application zones and/or regions by defining multiple availability zones. These zones can automatically scale up or down based on workloads and in addition there is a secondary region with the same setup of distributed zones for redundancy and ability to distribute the load during a partial or full outage situation [15], [16], [24].

FIGURE 3. **Clustered architecture of the application across multiple availability zones.**

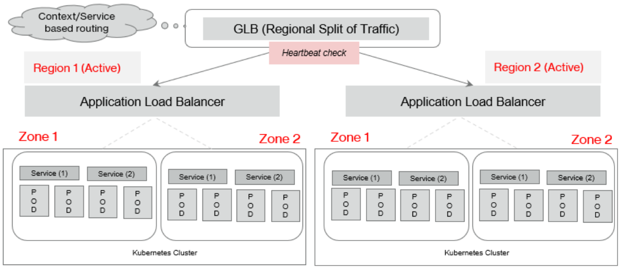


FIGURE 4. **Global load balancer routing traffic across zones for better utilization of the instances running in different data centers.**

Rather than reinventing the different solution components we have utilized existing tools and capabilities, integrated them together to accomplish this template architecture, as such Global Load Balancer (GLB) tool is integrated to constantly check the health of the application and perform a geo-location based routing, wherein the requests are routed to the nearest region [17], [18]. Fig. 4 below shows Global Load Balancer integrated with the application.

This can be accomplished by setting global load balancer rules to utilize user's IP address to determine the region with the closest proximity to serve the user requests. In addition, be context aware of the existence of multiple regions and define the routing tables according to the defined rules to accomplish the highest availability possible using the up and running resources [19].

C. STATELESS APPLICATION DESIGN

A stateless application design eliminates the complexity for the application to hold any data and session information. All the requests from the user are treated equally and no information of the prior requests or sessions is saved. An example of a stateless application will be a website hosting a static page that is served every time a request is received. The site does not save information of prior requests. This helps to load balance evenly across multiple servers or instances without being concerned about maintaining the sessions and data consistency. New instances of the application can be spun up to accommodate increase in traffic and traffic can be re-routed to an available instance in case of any failures. A stateful application on the other hand will present multiple different challenges related to synchronization of the state, redundancy and replication. The Fig. 5 below shows multiple pods are instantiated to support any spike

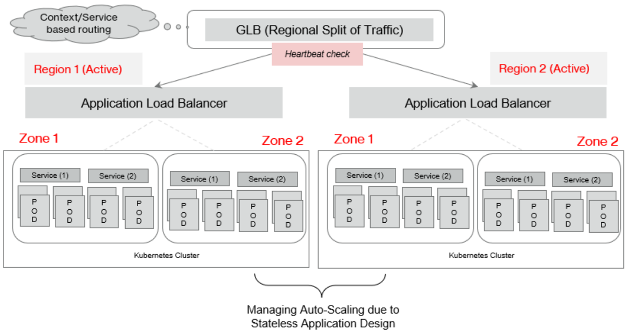


FIGURE 5. Autoscaling of the pods across the availability zone to manage any increase in traffic.

<table>
<tr><th colspan="2">HA Principles</th><th colspan="3">HA Mechanisms</th></tr>
<tr><td rowspan="2">Micro-services</td><td rowspan="2">Stateless Design</td><td>Redundancy Model</td><td>Redundancy Distribution</td><td>Overload Protection</td></tr>
<tr><td rowspan="3">▪ 2N: 1 active, 1 standby
▪ N+M: Multiple Active, 1+ Standby (No. of Standby < No. of Active)
▪ N-way: 1 active and X-Standby
▪ N-way-active: X-active
▪ No Redundancy: 1-active</td><td>▪ Geographic
▪ Cluster</td><td>▪ Autoscaling
▪ Load Balancing</td></tr>
<tr><td rowspan="2">Modular Design</td><td rowspan="2">Clustered Architecture</td><td colspan="2">Recovery Action</td></tr>
<tr><td colspan="2">▪ Failover/Switchover
▪ Restart
▪ Roll-back
▪ Roll-Forward</td></tr>
</table>

FIGURE 6. Table showing a synopsis of the high availability principles and mechanisms used to derive the template architecture.

in the traffic to the services hosted in stateless microservice architecture.

IV. RECOMMENDED APPLICATION ARCHITECTURE

Keeping the guiding principles explained above in consideration, this section recommends the system architecture for read intensive and write intensive database applications. In addition to the principles, the architecture leverages a subset of high availability mechanisms defined in the cloud taxonomy proposed in the Journal of Computer and Network Applications as summarized in Fig. 6 below. The mechanisms considered include Redundancy Model, Redundancy Distribution, Overload Protection and Recovery Action [1]. This paper combines the principles and mechanisms to recommend the template architecture for high availability.

A. RECOMMENDED ARCHITECTURE FOR READ INTENSIVE APPLICATIONS

For read intensive database applications, read and write operations are segregated. The application is deployed in primary and secondary region. The services are active in both regions and the global load balancer forwards the user requests to a given region based on close proximity of user's geographic location with the region. The primary database instance is the primary writer and the read replica is set up for read operations. The read replicas offload the read requests and this helps with the performance and stability of the primary database. The read requests are sent to both regions and write transactions are sent to the primary instance of

TABLE 1. High availability mechanisms – read intensive applications.

High Availability Mechanisms	Recommendation
Application Layer Redundancy Model	N-way-Active
Database Layer Redundancy Model	N-way- Active
Redundancy Distribution	Geographic Cluster
Overload Protection	Autoscaling Load Balancing
Fault Tolerance	Failover/Switchover

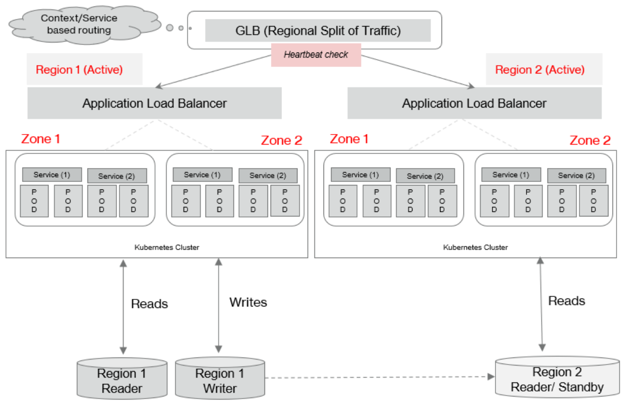


FIGURE 7. Recommended architecture for read intensive applications.

the writer database. In addition to routing the traffic, the global load balancer constantly performs the health check prior to sending the traffic to a particular service in a given region. The recommendation provided for a read intensive application is explained below and architecture is shown in Fig. 7.

1) APPLICATION LAYER REDUNDANCY: N-WAY-ACTIVE

All the instances play an active role and requests are routed in a load sharing manner. The same request can be fulfilled by any available instance. This is achieved by maintaining multiple instances across multi-zones

2) DATABASE LAYER REDUNDANCY: N-WAY-ACTIVE

For the database there is an active primary instance and there are two stand-by read instances. The data is replicated from the writer instance to the standby. The read instances serve as redundant elements though actively supporting the read transactions. The architecture has 1-active write in primary region and 1 – reader instance in each primary and secondary region. The reader will be promoted to a writer in case of any failure.

3) REDUNDANCY DISTRIBUTION: GEOGRAPHIC AND CLUSTER

The primary and secondary regions are in two different geographic regions. Also, the Kubernetes cluster is across multiple-zones within a given region

TABLE 2. High availability mechanisms – write intensive applications.

High Availability Mechanisms	Recommendation
Application Layer Redundancy Model	N-way-Active
Database Layer Redundancy Model	N+M
Redundancy Distribution	Geographic Cluster
Overload Protection	Autoscaling Load Balancing
Fault Tolerance	Failover/Switchover

4) OVERLOAD PROTECTION: AUTOSCALING AND LOAD BALANCING

The Kubernetes cluster is across multi-zones and the application is replicated in multiple pods. The number of pods can scale up in case of more traffic. At any point there is a minimum number maintained to manage the regular traffic. Similarly, the global load balancer is responsible to route the traffic based on geo-location and the application load balancer distributes the traffic within the region.

B. RECOMMENDED ARCHITECTURE FOR WRITE INTENSIVE APPLICATIONS

For write intensive database applications, redundancy is built by distributing the traffic between the two regions. In this use case, location-based data split is performed where the database is split and the data is stored based on the location and geographic region. This approach helps to optimize the access to data based on geographic proximity. Traffic originating from a particular geographic location is always diverted to the database belonging to that region. The global load balancer routes the traffic between the primary and secondary region. The application is deployed in both the region, whereby the services are available from both the region to serve any incoming requests. The recommendation for the write intensive applications is explained below and architecture is shown in Fig. 8.

1) APPLICATION LAYER REDUNDANCY: N-WAY-ACTIVE

Similar to read intensive recommendation, all the instances play an active role and requests are routed in a load sharing manner. The same request can be fulfilled by any available instance. This is achieved by maintaining multiple instances across multi-zones.

2) DATABASE LAYER REDUNDANCY: N+M

For the database there is an active primary instance in both the regions and there is a standby to support failover when the primary instance experiences any sort of service disruption. The data is split geo-location wise between the two primary instances for performance and scalability of the database. The requests originating from a given region will be directed to the database setup as primary for the given region. Both regions have a writer instance to manage the traffic originated from that region and in case of any failures, the traffic will failover to the standby available in the other region.

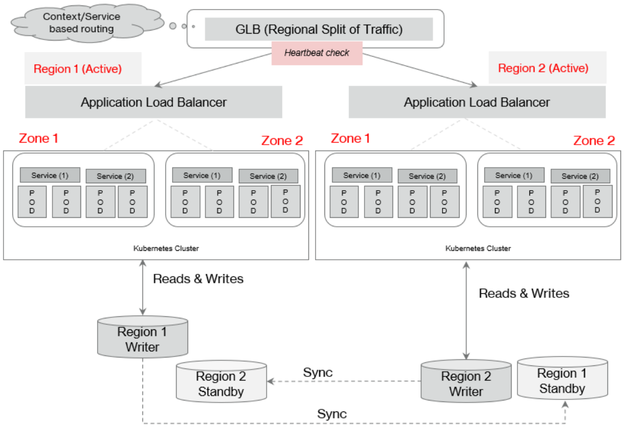


FIGURE 8. Recommended architecture for write intensive applications.

TABLE 3. Automatic database failover mechanisms for evaluation.

Options	Description
1	High Availability Proxy & PgBouncer
2	High Availability Proxy & Relation Database Service (RDS) Proxy

3) REDUNDANCY DISTRIBUTION: GEOGRAPHIC AND CLUSTER

The primary and secondary regions are in two different geographic regions. Also, the Kubernetes cluster is across multiple-zones within a given region.

4) OVERLOAD PROTECTION: AUTOSCALING AND LOAD BALANCING

The Kubernetes cluster is across multi-zones and the application is replicated in multiple pods. The number of pods can scale up in case of more traffic. At any point there is a minimum number maintained to manage the regular traffic. Similarly, the global load balancer is responsible to route the traffic based on geo-location and the application load balancer distributes the traffic within the region.

V. EXPERIMENTAL ANALYSIS TO BUILD DATABASE FAILOVER MECHANISM

With the pointers mentioned above, we can bring high availability, redundancy and scalability in the application architecture. The Global Load Balancer (GLB) and Application Load Balancer (ALB) helps with seamless switch over to the available resources and regions for the application layer. However, one of the biggest challenges is to perform automatic failover of the database. To resolve this challenge, two options are considered with a goal being to perform automated database failover and balance the traffic to the available database based on the continuous health checks [20], [21], [32]. The solution is also expected to reduce the overhead of managing multiple

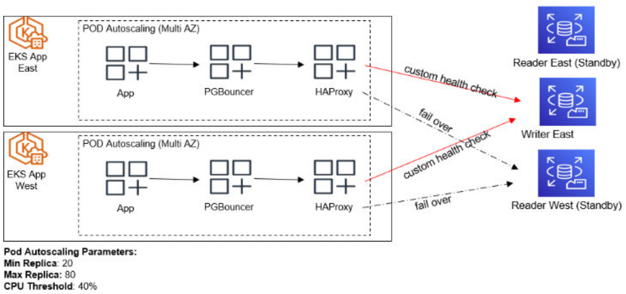


FIGURE 9. PgBouncer and high availability proxy (HAProxy) setup for database failover.

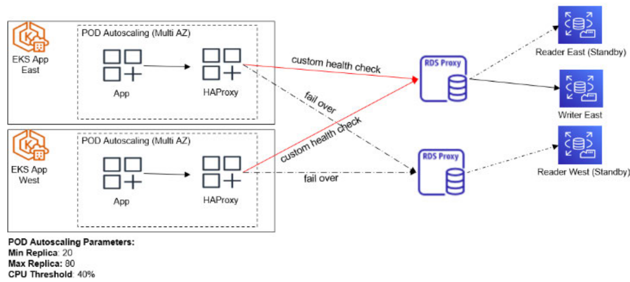


FIGURE 10. High availability proxy (HAProxy) and relational database service (RDS) proxy setup for database failover.

database connections and thereby improve performance and response time to the requests from the application.

The assessment was performed using the read intensive application and the acceptance criteria being performance and reliability to automatically failover to an available database during an outage or failure.

PgBouncer is a lightweight connection pooler for PostgreSQL and is designed to improve the database performance by reducing the overhead of connecting to the database. PgBouncer can manage multiple clients by reusing the pool of created connections [26]. This helps in reducing spin up of new connections and the idle connections are kept low, thereby increasing the performance, availability and reducing any possible database failures. In addition, for the database to be highly available High Availability Proxy's TCP load balancing capability are exercised along with PgBouncer.

High Availability Proxy (HAProxy) is an open-source reverse-proxy offering high availability and load balancing for TCP and HTTP based applications [22]. It performs continuous health checks on the database servers and routes the traffic to the most available server as mentioned in Fig. 9. In this solution, we use the leastconn algorithm for load balancing [23]. With the leastconn algorithm, the server with the lowest number of connections receives the connection. This algorithm is recommended for long lived connections.

Relational Database Service (RDS) Proxy is a fully-managed database proxy feature from Amazon and it helps improve the scalability and performance of the applications by managing and sharing the pool of database connections [27]. The setup for Relational Database Service (RDS) Proxy is shown in Fig. 10.

The High Availability Proxy (HAProxy) configuration file consists of four key sections: global, defaults, frontend and backend. We use the external-check command to call a custom shell script to implement a health check. In our case, we use this simply to ascertain if the backend server is the primary or secondary. We do this with a simple SQL command calling a PostgreSQL function called pg_is_in_recovery(), which returns a Boolean value to indicate whether it's the writer or the reader node. This check is needed as insurance to cover scenarios where in case of a failover, a reader node can be promoted without notice to a primary. Therefore, it's necessary to keep checking the status of a write replica to ensure that the application is connected to a writer node. Although the writer node on the west is always part of the High Availability Proxy (HAProxy) list of servers behind the endpoint, this custom health check ensures we do not use the writer node on the west and turn off the active flow of sessions against it. Similarly, in the event of a failover, the same script makes sure that the writer (which is now a reader node after a state change) gets its fair share of sessions to serve.

A validation setting at the PgBouncer level is the ''server_check_query'', which we use to make sure that the connections are not handed back to application servers before checking their validity. This is similar to the validate or validate on match feature present in some connection pool frameworks. This acts as insurance against handing off old dead sessions to the application [28].

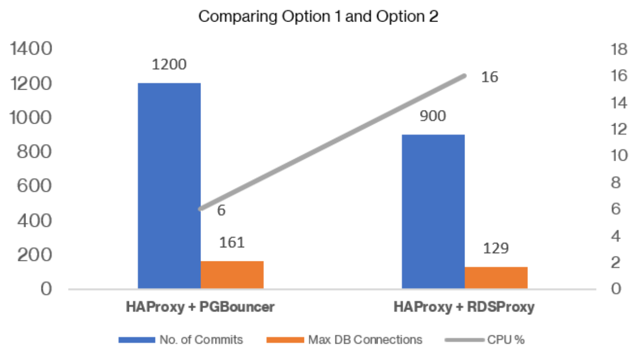


FIGURE 11. Commits and central processing unit (CPU) utilization captured for HAProxy/PgBouncer and HAProxy/Relational Database Service (RDS) Proxy.

HAProxy + PGBouncer					
Transactions	Expected SLA (sec)	Average (sec)	Passed	Error %	Throughput/sec
API 1	2	1.21	92813	0.00%	74.85529
API 2	2	1.46	77924	0.00%	62.83591
API 3	2	0.2	369991	0.00%	298.46601
API 4	2	0.2	371174	0.00%	299.42442

HAProxy + RDSProxy					
Transactions	Expected SLA (sec)	Average (sec)	Passed	Error %	Throughput/sec
API 1	2	1.84	63162	1.67%	49.71765
API 2	2	1.7	67400	0.88%	54.35313
API 3	2	0.2	357239	0.05%	288.17219
API 4	2	0.21	359463	0.00%	289.96809

FIGURE 12. Throughput and error rate captured for HAProxy/PgBouncer and HAProxy/ relational database service (RDS) Proxy.

Relational Database Service (RDS) Proxy sits between High Availability Proxy (HAProxy) and the database. As it is a managed service, its operations, scaling and maintenance overhead gets reduced but when it comes to being cloud agnostic, it has greater dependency on Amazon. It has a very limited set of configuration options which are available to the user for fine tuning and has dependency on Amazon support for troubleshooting and solution.

As shown in Fig. 11, High Availability Proxy (HAProxy) and PgBouncer combination had a comparatively less utilization of the processing resources even with a higher number of commits, compared to Relational Database Service (RDS) Proxy. Though, the max number of database connections pooled by PgBouncer were higher compared to Relational Database Service (RDS) Proxy. Dummy traffic was also created to evaluate the performance and the error rate.

It was observed that PgBouncer and High Availability Proxy (HAProxy) support fine-tuning of configuration and can scale based on the request. Wherein, Relational Database Service (RDS) Proxy does not support configuration tuning but takes care of scaling with its managed capabilities. As per Fig. 12, the application error percentage while using, Relational Database Service (RDS) Proxy has increased to 1.67% but with PgBouncer we were able to manage it at 0% error rate by fine tuning the configuration. On a whole, PgBouncer + High Availability Proxy (HAProxy) worked out well with respect to application response time and error percentage along with high availability and near zero downtime setup. This reduces the overall workload on the database and enables a productive usage of the aurora cluster. Now let's consider the impacts of not incorporating proposed PgBouncer and High Availability Proxy (HAProxy) components for traffic redirection.

A. ABSENCE OF PGBOUNCER

Processing time will significantly increase due to the distribution of the connection pooling tasks across application services where each service is not context aware of what is consumed on the other service. Higher number of connections would be sitting idle due to the lack of coordination across the application services that requires database connections. Also, support for services to restart or upgrade without dropping the client connections would be lost

B. ABSENCE OF HAPROXY

Support for automatic recognition of the database caused outages and gracefully redirecting traffic would be lost. Ability to perform zero downtime maintenance activities on the database would be significantly limited and would result in additional manual steps that would increase the overall recovery time and as a result not meet the high availability requirement of at least 99.999%

By adding PgBouncer and High Availability Proxy (HAProxy) services to the solution components we have saved manual processing time during the failover/failback

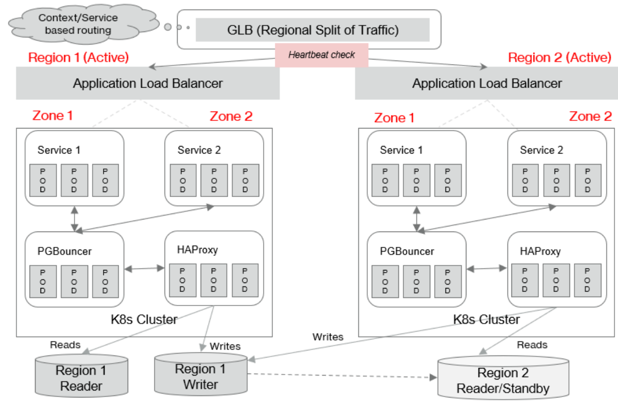


FIGURE 13. Read intensive database application architecture including PgBouncer and high availability proxy (HAProxy).

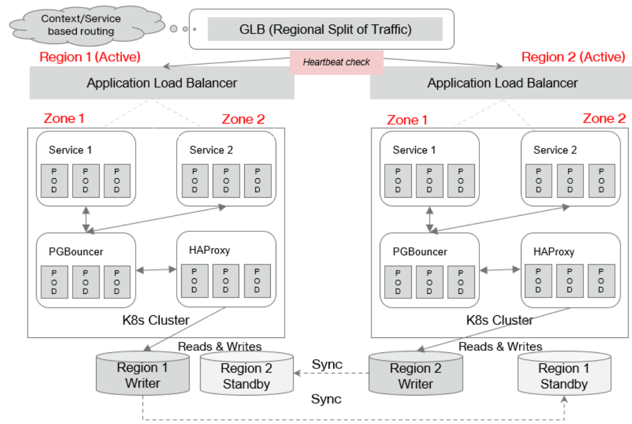


FIGURE 14. Write intensive database application architecture including PgBouncer and high availability proxy (HAProxy).

(Please refer to Table 6 in Section (V) for more details on how this been calculated and concluded)

VI. FAILOVER SCENARIOS

We included PgBouncer and High Availability Proxy (HAProxy) services to the solution as to perform automated application and database failover scenarios [29], [30], [31]. For the purposes of this exercise, we have defined our Recovery Time Objective (RTO) to be "30 seconds" and Recovery Point Objective (RPO) time to be "5 minutes" to accomplish the high availability percentage of 99.999%. RTO is the maximum reasonable time for the service to be interrupted and RPO is the maximum time allowed between the two backups [33], [34]. Reference the following architecture diagram in Fig. 13 and Fig. 14 for read and write intensive database applications:

A. ENVIRONMENT SETUP

We have the used the following system components to setup the study environment, however these are not compulsory and the environment can be setup in a different way. Two regions are used for this exercise where Region (1) is the

TABLE 4. Failover scenarios considered for assessment.

Scenarios	Read - Intensive	Write - Intensive
In Region Failover – Partial Application Failure	Y	Y
In Region Failover – Full Application Failure	Y	Y
Full Region Failover	Y	Y

TABLE 5. Application in - region failover evaluation.

		In Region – Partial Failover	In Region – Full Failover
Application Failover	Application Layer	The application partially fails at the service level	The application fully fails over at the GLB level and routes the traffic across regions
	Approach	Failover triggered when service is down in both zones within primary region	
	Failover	GLB pre-defined rules including constant service level health checks determine when to switch traffic	
	Failback	GLB pre-defined rules to fail back based on heartbeat signals	
	Failure Detection & Failover SLA	*SLA = (Check Alive Interval + Response timeout) * Failure Threshold* Using the following parameters: Check Alive Interval: Every 10 secs Response Timeout: 4 secs Failure Threshold: 2 tries *(10 + 4) * 2 = 28 secs*	
	Recovery Detection & Failback SLA	*SLA = (Check Alive Interval + Response timeout) * Success Threshold* Using the following parameters: Check Alive Interval: Every 10 secs Response Timeout: 4 secs Success Threshold: 2 tries *(10 + 4) * 2 = 28 secs*	
	RTO	~ 30 secs	~ 30 secs

TABLE 6. Database failover evaluation.

		Read Intensive	Write Intensive
Database Failover	Reader	Available	NA
	Writer	Available	Available
	Standby	Available	Available
	PgBouncer	Connection Pooling Controller – Ensures that the databases are set to maintain the configured pool of connections and reduces the pressure on the database instance	
	High Availability Proxy (HAProxy)	Heart beats to redirect traffic	
	Database Layer (Reader Failure)	Failure to secondary region in case of primary readers failure	NA – There is no read replicas setup
	Database Layer (Writer Failure)	Failover to secondary region in case of writer failure	
	Replication SLA	AWS Writer/AWS Readers ~ 1 sec	Golden Gate Replication ~ 1 sec
	High Availability Proxy (HAProxy) SLA	SLA = Max (Timeouts) + Check Alive Interval Using the following parameters: Check Alive Interval: Every 10 secs Client timeout: 150 secs Connect timeout: 150 secs Server timeout: 150 secs HTTP timeout: 150 secs	
	RPO	~ 5 minutes	~ 5 minutes

primary region while Region (2) is the secondary and for each region two zones are assumed as Zone (1) and Zone (2). Application deployment is required for each zone and for each region. There is a Global load balancer (GLB) to balance traffic across regions. Also, a Kubernetes cluster per region as primary and secondary. The Kubernetes worker nodes are setup for each cluster/region with a minimum of (3) nodes across multiple availability zones. There is an Application Load Balancer (ALB) configured to balance the traffic across services for each region. The application deployed consists of Service (1) and Service (2) for executing failover scenarios. The exercise utilized the benefits of stateless microservices along with the underlying Kubernetes deployment capabilities to run each service with two or more pods per service and these pods can auto scale as needed based on the workload. The auto scaling is set up based on workload. PgBouncer service is used part of the application services for connection pooling combined with a lightweight connection pool handler PGBouncer to reduce the processing and memory pressure and downtime at the database layer.

High Availability Proxy (HAProxy) service is used as part of the application services for heart beat checks and traffic redirections. The High Availability Proxy (HAProxy) service on Region(1) primary is configured with a database cluster read endpoints on the same region with maximum weightage to increase the connection priority to its respective regions endpoint and to reduce the network latency. A Database cluster for read intensive consists of two nodes for Region (1) where node (1) acts as a writer and node (2) acts as a reader. The node on Region(2) acts as a reader to handle read transactions and will be promoted as a writer in case of any failures. The database cluster for write intensive includes two nodes for each region where node (1) acts as a writer and node (2) acts as a standby. Health check script is configured as part of the High Availability Proxy (HAProxy) service. The health check script can be an external check command with a custom shell script that indicates the status of the database writer, reader/standby nodes. For example, database failover checks and PostgreSQL in recovery APIs can be used or any other APIs of choice depending on the database used. Also, for this exercise the method of replication followed is Amazon aurora replication for read intensive application and golden gate replication for write intensive application.

TABLE 7. Application full region failover evaluation.

		Full Region Failover
Application Database Failover	Application Layer Database Layer	Full application failure would failover at the Global Load Balancer level and route the traffic to the secondary region. Both Application and Database will be impacted
	Approach	Failover is triggered when the complete region is unavailable
	Failover	Global load balancer predefined rules determine when to switch traffic across
	Failback	Global load balancer predefined rules to failback based on the heartbeat signals
	Failure Detection/ Failover SLA	*SLA = Application Failover SLA + Database Failover SLA* Application Failover SLA: Failure Detection & Failover SLA (Ref. Table V) **~ 30 seconds** Database Failover SLA: HAProxy SLA (Ref Table VI) **~ 3 mins**
	Failback SLA	*SLA = Application Failback SLA + Database Failback SLA* Application Failback SLA: Recovery Detection & Failback SLA (Ref. Table V) **~ 30 seconds** Database Failback SLA: HAProxy SLA (Ref Table VI) **~ 3 mins**
	RTO	~ 4 minutes

B. FAILOVER SCENARIOS CONSIDERED

See Table 4.

C. ASSUMPTIONS

The write intensive operational model stated here is based on the specific region data split principle where each region has its own set of data sets. These data sets are not required to be accessible across regions. Regional data split is achieved by separating database repositories that holds the specific data sets for each region. The assessment is done on the prescribed application architecture and does not include scenarios for write intensive applications without region split and read intensive with region split. PgBouncer and High Availability Proxy (HAProxy) though appear to be single points of failure as per the Fig. 13 and Fig. 14, however the risk can be mitigated with further redundancy for both services across regions and additional traffic rules for routing. In an event High Availability Proxy (HAProxy) and PgBouncer has to fail, in order to avoid a complete outage or failed requests from services, it is assumed that there exists circuit breaker that will help notify the global load balancer to failover the traffic to the secondary region. Presence of application logic and business rules to ensure that the application is shutdown gracefully to avoid any data losses in case of a failover. In the absence of a graceful application shutdown, data in the application and database memory will be lost. However, data stored or persisted at the time of failover will be saved.

D. RESULTS

To trigger in-region failover when there is a partial application failure, Service (1) is deleted in the primary region. Similarly, for a full application layer failure, both Service (1) and Service (2) are deleted in the primary region. For a full region failover, the entire region is disconnected from the global load balancer.

Observations recorded are in tables Table 5, Table 6, Table 7. Table 5 captures in-region full and partial failover with no impact to the database layer. Table 6 captures the database failover behavior and provides SLA for High Availability Proxy (HAProxy) and PgBouncer. Finally, Table 7 captures the scenario for full region failure and provides the SLAs recorded when both application and database are down in the primary region.

VII. CONCLUSION

A highly available application is key for the success of the business therefore the focus is to evaluate different solutions to achieve high availability or near zero downtown using cloud native architecture. Database applications migrating to cloud have guidance to achieve high availability, however building resiliency and performing automated database failover can be challenging. In our solution we have profiled the database applications as read intensive and write intensive and then evaluated the recommended architecture for high availability. Microservices based architecture, modular and stateless design are an integral part of the design principles considered in this study. The recommended application architecture for high availability has a geographically distributed application model with multiple zones that support within region failures. Application and database redundancy is built using N-way Active/Active or Active/Passive with auto scaling. Global and Application load balancer to manage traffic across and within region. To build redundancy at the application layer, services are deployed as replicas across region with auto scaling parameters and database replication using either regional or non-regional split approach depending upon the application and business use case. PgBouncer and High Availability Proxy (HAProxy) services are incorporated for optimized connection pooling and health checks for routing traffic for failover and failback.

These principles and recommendations enabled us to achieve availability level at 99.999% or a permissible downtime of 5 minutes, 16 seconds. In addition, leveraging PgBouncer and High Availability Proxy (HAProxy) enabled failover of the database with a click. In this exercise, planned outages or maintenance activities are executed and there is opportunity to further experiment with unplanned outages as there may be other factors that are necessarily not covered in this experiment. An uninterrupted service is a non-functional requirement and the solution can vary based on the size of the application, the amount of data being processed and its footprint. However, the principles, architecture and failover mechanisms prescribed in this paper can be a starting point to build resilient applications. The blueprint architecture and failover techniques recommended in this paper can be customized as needed to match an application's unique needs and it is important to factor the cost involved and the business case that is being addressed to assess the high availability requirements for a given application.

ACKNOWLEDGMENT

The authors would like to thank the support and contribution of these team members toward the study and implementation of best practices for high availability and near zero downtime experience.

- DevOps Engineers: Rubankumar Sathyamoorthy and Yaswanth Nadella
- Application Development Engineers: Deepak Kumar, Aris Fernandez and Archana Dodanari
- Verizon India Partners: Saravanan Ramasamy, Saranya Kumaraguruparan, Ritu Sharma, Suvarna Thatiparthi and Thejesh Kanuparthi.

REFERENCES

[1] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *J. Netw. Comput. Appl.*, vol. 60, pp. 54–67, Jan. 2016, doi: 10.1016/j.jnca.2015.11.014.

[2] M. Toeroe and F. Tam, "Introduction to service availability," in *Service Availability Principles and Practice*. Hoboken, NJ, USA: Wiley, 2012.

[3] H. Rohani and A. K. Roosta. *Calculating Total System Availability*. Whitepaper. Accessed: Apr. 15, 2023. [Online]. Available: http://d1.awsstatic.com/whitepapers/architecture/CalculatingTotalSystemAvailability.pdf

[4] R Publishing. *Availability and the Different Ways to Calculate it,' Weibull.Com—Free Data Analysis and Modeling Resources for Reliability Engineering*. Accessed: Mar. 10, 2023. [Online]. Available: https://www.weibull.com/hotwire/issue79/relbasics79.htm

[5] E. Bauer and R. Adams, "Service reliability and service availability," in *Reliability and Availability of Cloud Computing*. Hoboken, NJ, USA: Wiley-IEEE Press, 2012.

[6] P. T. Endo, G. L. Santos, D. Rosendo, D. M. Gomes, A. Moreira, J. Kelner, D. Sadok, G. E. Gonçalves, and M. Mahloo, "Minimizing and managing cloud failures," *Computer*, vol. 50, no. 11, pp. 86–90, Nov. 2017, doi: 10.1109/mc.2017.4041358.

[7] W. T. Millward. *The 15 Biggest Cloud Outages of 2022*. CRN. Accessed: May 10, 2023. [Online]. Available: https://www.crn.com/news/cloud/the-15-biggest-cloud-outages-of-2022

[8] C. Pham, P. Cao, Z. Kalbarczyk, and R. K. Iyer, "Toward a high availability cloud: Techniques and challenges," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN)*, Jun. 2012, pp. 1–6, doi: 10.1109/dsnw.2012.6264687.

[9] Z. Kerravala, V. Jain, M. Stern, B. Herzberg, S. Salamone. (2022). *Lessons Learned From the Top Cloud Outages of 2022*. Network Computing. [Online]. Available: https://www.networkcomputing.com/cloud-infrastructure/lessons-learned-top-cloud-outages-2022

[10] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 16–21, Sep. 2017, doi: 10.1109/mcc.2017.4250939.

[11] G. Reese, *Cloud Application Architectures: Building Applications and Infrastructure in the Cloud*. Beijing, China: O'Reilly, 2010.

[12] Y. Izrailevsky and C. Bell, "Cloud reliability," *IEEE Cloud Comput.*, vol. 5, no. 3, pp. 39–44, May 2018, doi: 10.1109/mcc.2018.032591615.

[13] S. G. Haugeland, P. H. Nguyen, H. Song, and F. Chauvel, "Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps," in *Proc. 47th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Sep. 2021, pp. 170–177, doi: 10.1109/seaa53835.2021.00030.

[14] C. Richardson. *What are microservices?* Accessed: May 10, 2023. [Online]. Available: https://microservices.io/

[15] H. Sun, J. J. Han, and H. Levendel, "A generic availability model for clustered computing systems," in *Proc. Pacific Rim Int. Symp. Dependable Comput.*, 2001, pp. 241–248, doi: 10.1109/prdc.2001.992704.

[16] G. Sayfan, "High-availability best practices," in *Mastering Kubernetes: Automating Container Deployment and Management*. Birmingham, U.K.: Packt Publishing, 2017.

[17] S. K. Mishra, B. Sahoo, and P. P. Parida, "Load balancing in cloud computing: A big picture," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 32, no. 2, pp. 149–158, Feb. 2020, doi: 10.1016/j.jksuci.2018.01.003.

[18] S. Meera and K. Valarmathi, "Load balancing techniques in cloud environment—A big picture analysis," in *Proc. 1st Int. Conf. Comput. Sci. Technol. (ICCST)*, Nov. 2022, pp. 307–310, doi: 10.1109/iccst55948.2022.10040387.

[19] (2023). *Load Balancer—Amazon Elastic Load Balancer (ELB)—AWS*. [Online]. Available: https://aws.amazon.com/elasticloadbalancing

[20] M.-L. Yin, "Assessing availability impact caused by switchover in database failover," in *Proc. Annu. Rel. Maintainability Symp.*, Jan. 2009, pp. 401–406, doi: 10.1109/rams.2009.4914710.

[21] I. D. Addo, S. I. Ahamed, and W. C. Chu, "A reference architecture for high-availability automatic failover between PaaS cloud providers," in *Proc. Int. Conf. Trustworthy Syst. Their Appl.*, Jun. 2014, pp. 14–21, doi: 10.1109/tsa.2014.12.

[22] J. E. C. de la Cruz and Ing. C. A. R. Goyzueta, "Design of a high availability system with HAProxy and domain name service for Web services," in *Proc. IEEE 24th Int. Conf. Electron., Electr. Eng. Comput. (INTERCON)*, Aug. 2017, pp. 1–4, doi: 10.1109/intercon.2017.8079712.

[23] A. B. Prasetijo, E. D. Widianto, and E. T. Hidayatullah, "Performance comparisons of web server load balancing algorithms on HAProxy and heartbeat," in *Proc. 3rd Int. Conf. Inf. Technol., Comput., Electr. Eng. (ICITACEE)*, Oct. 2016, pp. 393–396, doi: 10.1109/icitacee.2016.7892478.

[24] A. Garg and S. Bagga, "An autonomic approach for fault tolerance using scaling, replication and monitoring in cloud computing," in *Proc. IEEE 3rd Int. Conf. MOOCs, Innov. Technol. Educ. (MITE)*, Oct. 2015, pp. 129–134, doi: 10.1109/mite.2015.7375302.

[25] HAProxy. (2023). *The Reliable, High Performance TCP/HTTP Load Balancer*. [Online]. Available: https://www.haproxy.org/

[26] PgBouncer. (2023). *News*. [Online]. Available: https://www.pgbouncer.org/

[27] D. Kopitz and B. Marks. *RDS: The Radio Data System*. Amazon. Accessed: May 20, 2023. [Online]. Available: https://aws.amazon.com/rds/proxy/

[28] L. Q. Ha, J. Xie, D. Millington, and A. Waniss, "Comparative performance analysis of PostgreSQL high availability database clusters through containment," *IJARCCE*, vol. 4, no. 12, pp. 526–533, Dec. 2015, doi: 10.17148/ijarcce.2015.412150.

[29] S. Sengupta and K. M. Annervaz, "Multi-site data distribution for disaster recovery—A planning framework," *Future Gener. Comput. Syst.*, vol. 41, pp. 53–64, Dec. 2014, doi: 10.1016/j.future.2014.07.007.

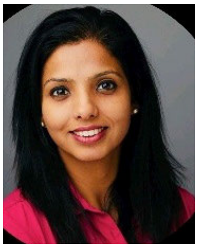
[30] Y. Ping, K. Bo, L. Jinping, and L. Mengxia, "Remote disaster recovery system architecture based on database replication technology," in *Proc. Int. Conf. Comput. Commun. Technol. Agricult. Eng.*, Jun. 2010, pp. 254–257, doi: 10.1109/cctae.2010.5544352.

[31] Z. Huang, J. Chen, Y. Lin, P. You, and Y. Peng, "Minimizing data redundancy for high reliable cloud storage systems," *Comput. Netw.*, vol. 81, pp. 164–177, Apr. 2015, doi: 10.1016/j.comnet.2015.02.013.

[32] D. Singh, J. Singh, and A. Chhabra, "High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms," in *Proc. Int. Conf. Commun. Syst. Netw. Technol.*, May 2012, pp. 698–703, doi: 10.1109/csnt.2012.155.

[33] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing," in *Proc. 14th Int. Conf. Extending Database Technol.*, Mar. 2011, pp. 13–53, doi: 10.1145/1951365.1951432.

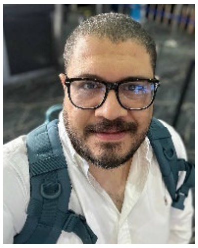
[34] A. Lenk and S. Tai, "Cloud standby: Disaster recovery of distributed systems in the cloud," in *Proc. Adv. Inf. Syst. Eng.*, pp. 32–46, 2014, doi: 10.1007/978-3-662-44879-3_3.



ANTRA MALHOTRA (Member, IEEE) was born in New Delhi, India. She received the master's degree in business and finance from Mumbai University, Mumbai, India, in 2006, and the master's degree in computer science from the Georgia Institute of Technology, Atlanta, GA, USA, in 2022.

In 2023, she was appointed as a Guest Faculty with the Hillsborough Community College, ICCE, for teaching cloud concepts and technologies. She is currently with the Verizon Network Systems Team, as a Senior Manager of the Systems Engineering, Temple Terrace, FL, USA. She is responsible for the application development and software delivery for the two big data platforms that provide intelligent location services, product availability and business intelligence for Verizon's Enterprise and Consumer business groups. Prior to joining Verizon, she was with Hutchison 3G UK, Mumbai, and Acclaris (Product based company in Tampa, FL, USA). Her research interests include cloud technologies, application stability, and big data management.

Mrs. Malhotra has earned the AWS Solution Architect Certification.



AMR ELSAYED was born in Cairo, Egypt. He is a technology geek. He currently holds the position of the Principal Engineer-System Architecture with Verizon, Temple Terrace, FL, USA. He plays a pivotal role in providing innovative solutions across Verizon's Network Systems. Before joining Verizon, he was with the IBM Clients Innovation Center, where he honed his skills and expertise in the technology field. His diverse talents and dedications make him a valuable asset in the technology industry and a source of inspiration for aspiring writers.

RANDOLPH TORRES was born in Miami, FL, USA. He received the Bachelors of Arts in Science and Computer Science degree from Florida International University, USA. He is currently with Verizon Network Systems, Temple Terrace, FL, USA, and leads the architecture and technology strategy for the shared platforms organization. His responsibilities range from AI/ML modeling to complex architectures relating to high performance computing, data integration, and fault resilience. He has contributed to Open Config and holds many patents for network systems in production today at Verizon.

SRINIVAS VENKATRAMAN was born in Chennai, India. He received the Master of Engineering degree from the Indian Institute of Science in Metallurgy. He is currently the Director of the Systems Engineering, leading the Shared Platform Portfolio for Verizon, Basking Ridge, NJ, USA. In his current role, he manages suite of enterprise applications and tools that facilitate address and location management, product availability intelligence, common system of engagement, data warehouse, and large-scale workflow platforms. He is passionate about technology and leads the technology recommendation group across network systems.