aws FOR FINANCIAL SERVICES

DTCC

Designing Highly Resilient Financial Services Applications

Based on a reference implementation developed for The Depository Trust & Clearing Corporation (DTCC)

First published November 20, 2023

Last updated November 20, 2023

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.

Contents

Abstract and introduction

Abstract

This paper focuses on how to apply the Depository Trust & Clearing Corporation's (DTCC) resilience principles to real world, mission critical systems. It provides specific architectural guidance to help customers increase the resilience of their applications. It also provides a sample reference implementation of an equities trade matching and settlement application using resiliency features implemented with AWS services. It concludes with an end-to-end, working reference implementation of the guidance available in GitHub.

Introduction

For over 45 years, the Depository Trust & Clearing Corporation (DTCC) has played a pivotal role in protecting and supporting the growth of the global financial markets, tackling the industry's biggest operational challenges collaboratively, while processing millions of securities transactions every day. DTCC, in partnership with the financial services industry, has navigated extreme events, evolving business continuity planning needs and data center redundancy expectations, and increased transaction processing capacity requirements for critical products and services.

In this whitepaper, we will dive deep into the strategy and approach for delivering resilient cloud applications. AWS and DTCC have partnered to collaborate on a prototype to realize their resiliency principles using a simulated business process. Deliverables from this work included a framework, a set of best practices, and a reference implementation for resilient applications informed by common requirements in the financial services industry.

As stated in the 2022 whitepaper, *The Power of Technology Resilience: A Framework for the Industry*, DTCC understands the importance of its position as a critical infrastructure and service provider for the global capital markets. In fact, DTCC follows stringent out-of-region (DR) recovery and resumption requirements for all of its critical services, meeting the regulatory required two-hour recovery time objective (RTO), and a data recovery point objective (RPO) of merely 30 seconds cross region. DTCC's out-of-region recovery locations must be hundreds of miles away from the primary data center, and on a completely separate power transmission interconnections with a separate physical telecommunications path. Regulators and supervisors are laser-focused on ensuring DTCC tests and proves its resilience capabilities, consistently raising expectations for how to implement its most critical systems and software.

The AWS partnership with DTCC has been strong for many years. At the 2018 AWS Summit in New York, Robert Palatnick, Managing Director and Head of Technology Research and Innovation, described how DTCC's AWS footprint began with the migration of analytics workloads to provide faster performance and cost reductions to the market.

Goals and outcomes

In 2022, AWS and DTCC extended their partnership to develop a reference implementation using AWS services for DTCC's most critical ("tier one") applications to meet the following goals:

- Identify a starting point for delivering resiliency in a public cloud environment that can be used as a foundation for any application
- Create architecture patterns and software assets that demonstrate DTCC's resiliency principles
- Embed resiliency into applications through the development and consumption of reusable components and capabilities
- Enable applications to operationally rotate between data center regions and run in each region for an extended period of time
- Solution solves for planned (scheduled rotation) and unplanned (disaster recovery) events

Outcomes:

- Demonstrate an AWS architecture for a stream-based message processing applications for the following functional use cases:
    - Trade Matching Reference Application
    - Settlement Reference Application
- Demonstrate that the proposed prototype architecture meets DTCC's non-functional requirements for resiliency, both in region and out of region
- Infrastructure-as-code to configure AWS services to host the workload in two AWS regions
- Application code for two representative sample applications which communicate with each other as part of a business process
- Automated runbooks to orchestrate the rotation of each individual application from the current active region to an alternate region. Each application rotates independently and must include replaying transactions from a persistent store and reconciling differences in data state among stages of the business process
- A dashboard to observe the availability of application components and completeness of data replication during a planned or unplanned event

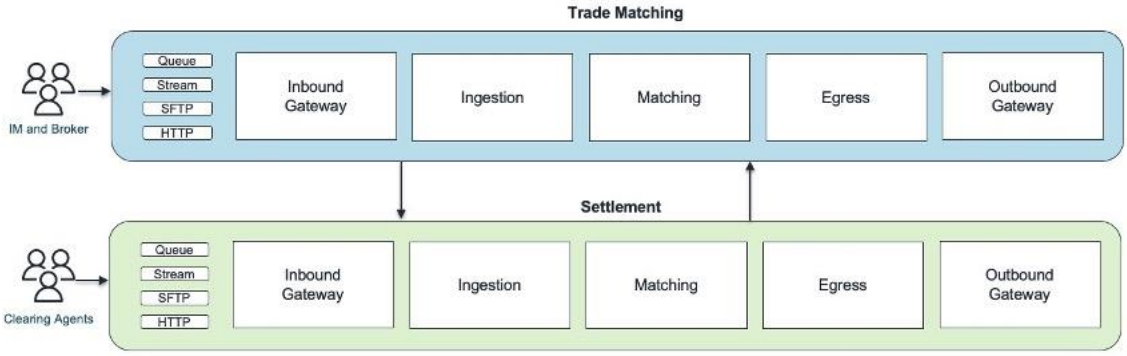
This paper will describe the key considerations when designing applications to be highly resilient and the rationale for the specific choices AWS and DTCC made in this reference implementation.

Trade matching and settlement reference applications

With the goals and objectives defined, we identified trade matching and settlement reference applications to be implemented for the prototype. The example applications should be complex enough to approximate real world applications yet simple enough for technical and risk personnel to reason over. Like real world applications, they should be interconnected and reliant on constant communication with one another to complete their operations. The two applications chosen represent a basic trade matching and settlement process.

Succinctly, the trade matching application receives blocks of trades from both brokers and investment managers. It then matches the trades received from brokers with those received from investment managers, and once matched, the trades can be sent to the settlement application.

The settlement application receives the trades from the trade matching application and ensures the seller's account is credited, the buyer's account is debited, and fees are paid to brokers involved. Once a trade is settled, it is sent back to the trade matching application. The trade matching application marks the trade as settled and sends notifications to the investment manager and the broker who had submitted the trade. The high-level design and interactions of these two applications is depicted in the following image:



High level design for trade matching and settlement example application interactions

Resilient Financial Services Applications

Applications which automate business processes like equities trading and settlement are highly complex. The process requires multiple steps which are implemented by separate subsystems. The state of the system at any point is a function of the state of each individual subsystem and its data store. To prove that the overall system is resilient in the face of various failure modes requires careful planning, design, and testing. This section describes DTCC's principles for delivering a resilient solution. These principles will be used to illustrate a design that delivers loosely coupled applications that can independently operate in either region, demonstrate granular failure and recovery boundaries, and allow for testing production applications more proactively by relocating between alternate regions.

In *The Power of Technology Resilience*, DTCC enumerates technology resilience across four categories of principles:

- **Regional Availability**: Architecture must be designed for redundancy with auto-correct capabilities for each component within and across local sites by leveraging multiple instances of data, compute, and networks. Applications and infrastructure need to perform under all circumstances and require targeted planning for capacity needs.
- **Design Resilient Applications**: Applications should be designed to detect internal and external failures and incorporate capabilities to recover from such failures, safely leveraging automation whenever possible. Resilient applications are also designed to be independent of other applications to help isolate data and compute failures. Applications should also be capable of having their workloads rotated across multiple data centers.

- **Leverage Out-of-Region Recovery**: Applications should be able to recover from disruptions and incidents at an alternate region to protect from local region failure scenarios affecting service availability. Ensure capabilities are augmented to maintain data consistency across regions, such as data reconciliation tools to identify and remediate gaps.
- **Resilience Success**: All solution designs require adequate validation processes so that each critical business service can determine its health, and resilience success can be verified upon recovery by leveraging key performance indicators and automated responses when possible. Controls should be created to help prevent the corruption and/or destruction of production or reference data, source code and configuration data.

For the purpose of this effort, this paper mainly focuses on implementing solutions that satisfy the Design Resilient Applications and Out-of-Region Recovery categories of principles for applications running in an AWS environment. In addition, it will also provide guidance on achieving in-region availability, failure detection, automated remediation, validation, and other key aspects of delivering fully resilient applications.

Out-of-region recovery patterns (planned and un-planned events)

For out-of-region recovery in AWS, customers must choose a disaster recovery (DR) pattern based on budget, time-to-recover goals, data loss tolerance of the system, and dependencies on separate internal and external systems (e.g., AWS control plane APIs). This design must account for both a planned event, which is the migration of an application from the active region to an alternate region, or what DTCC refers to as “application rotation,” and an unplanned event, which requires an out-of-region recovery due to an abrupt and prolonged inability to operate in the primary region. During a planned event when the system is healthy, the process will execute a graceful shutdown and migration of data. During an unplanned event, data loss may occur depending on the criticality and extent of the availability event, and the priority is restoring service as soon as possible. Data loss or inconsistency are addressed later through compensating transactions and processes which are outside of the system.

DTCC’s requirements for the DR mechanism of critical applications include a recovery time objective (RTO) of less than two hours and a recovery point objective (RPO) of less than 30 seconds. This means that if a significant regional availability event impacted the trade matching or settlement applications, DTCC must demonstrate to internal and external risk management entities that they can recover service in an alternate region within that time and that messages older than 30 seconds will be processed after failover to the alternate region.

There are four recommended patterns to implement out-of-region recovery:

1. **Active / active** - Equivalent infrastructure is deployed in both regions and both regions handle business traffic simultaneously. The RTO and RPO can both be near zero, but cost and complexity to avoid data inconsistency between the regions are higher. Some applications may need to be redesigned to support this pattern because they assume only one authoritative data store is active at any one time.

2. **Active / standby** - Equivalent infrastructure is deployed in both regions, however only one region handles business traffic, while the other region is in a warm standby state, ready to take over traffic on a short notice. RTO can be in minutes and RPO in seconds and since only one authoritative data store is active at a time, applications often don't need to be redesigned.
3. **Pilot light** - Full infrastructure is deployed in the primary region which handles business traffic, and a scaled down version of infrastructure is deployed in the secondary region, which needs to be primed up to match the infrastructure deployed in the primary region before traffic can be routed to it. RTO can be in tens of minutes. RPO varies based on the mechanism used to synchronize data changes to the alternate region.
4. **Backup and restore** - The primary region handles traffic and backs up data stores to another region, and in case of disaster, the infrastructure and application code is deployed to the alternate region. Data is restored from the backup data before traffic is routed to the secondary region.

Critical financial services applications typically require low RTO and RPO, but also have performance requirements which cannot tolerate the latency of a geographically separated active / active architecture. The benefits of an active / active architecture often do not justify the cost of redesigning the application to support multiple active data stores accepting updates. The Pilot Light and Backup and Restore patterns were dismissed due to not meeting the RTO and/or RPO requirements. For these reasons, this paper and the accompanying reference implementation will focus on an active / standby pattern.

Resilient System Design

This section describes specific design recommendations based on the resilience principles enumerated earlier and the goal to demonstrate a reliable, repeatable multi-region disaster recovery pattern for applications running in AWS.

Key capabilities of a resilient system

A system which is resilient even during large scale availability events is more than a function of the technical design of its business function. To be resilient, the system must include automated orchestration, monitoring, and testing capabilities. These are described below.

Orchestration

Orchestration is consistently and accurately creating and configuring system components during a planned or unplanned recovery event. This should be automated in code with clear feedback to site reliability engineers (SREs) on the state and success of each step of the recovery process. Consider the alternative: a documented runbook of steps which the SRE should follow during a recovery event. Small errors in the order, completeness, and verification of each step could introduce new issues in the target region. This could leave the entire system in an inconsistent state and unable to restore service in either region.

While the decision to execute this orchestrated failover could be automated based on metrics of the system's performance, this decision usually requires human judgement to weigh the risks of continuing to troubleshoot the resilience event in the current region versus the risks of invoking the failover orchestration. If the orchestration mechanism is automated and has been rehearsed during many recovery exercises and planned rotation events, the human decision-makers should have high confidence in the amount of recovery time and data loss to expect (ideally zero). This makes the go / no-go decision

to push the failover button much less risky, leading to a shorter decision timeframe, and therefore a faster recovery time.

While the automated failover orchestration runbook for the planned and unplanned scenarios may share many components, they will differ in some subtle ways. In the planned scenario, all systems are functioning normally, so the runbook can shutdown components in an orderly fashion and wait for in-flight transactions to flow through the system. The data associated with those in-flight transactions will be replicated to the secondary region before allowing new transactions to be processed. During a significant availability event which affects the system's ability to complete in-flight transactions and replicate data, there is no reason for the runbook to wait for that work to complete. It will skip those steps and focus on restoring service in the second region as soon as possible with the expectation that some data may be lost.

It is also worth noting that the planned event runbook might sometimes be appropriate even during an unplanned event. For example, consider a scenario where performance monitoring shows individual application components like the trade matching microservice or database replication working, but exceeding performance targets. The system is still working, but just slower than expected. The graceful failover in the planned runbook may be appropriate in order to minimize data loss while restoring service in the new region.

The orchestration runbooks for the reference applications are implemented as an AWS Systems Manager Automation document. Each step in the runbook invokes a small program which uses the AWS Software Development Kit (SDK) for Python (boto3) to take appropriate actions to check and configure the state of the services required by the application. Examine the automation document source in the repository under infrastructure/apps/common/rotation/ or the AWS Systems Manager Automation console (after deploying the reference implementation to your AWS Account) for a detailed example of automated orchestration of a multi-region, active / standby DR pattern.

Monitoring

The second important capability for a resilient application is to monitor critical infrastructure and application metrics, display the information in a dashboard which human decision-makers can easily reason over, and raise alerts when the metrics breach applicable thresholds. This allows SREs to decide what action to take quickly and confidently.

It is unlikely that the initial design of what to display and when to alarm will be the optimal one. Rather, a continuous improvement mechanism like the AWS Well Architected Framework's Correction of Error (COE) process should be implemented to continually refine the dashboard and alarms based on learnings from actual events.

The reference implementation monitors the infrastructure at three levels. First, it uses the AWS Health Check API to validate that the AWS services used by the application are operating normally. In the rare event that a problem with the application in one region is correlated with an availability event with one or more services it depends on in the second region, failover may introduce more risk than remaining in the current region. Surfacing as much actionable detail to the SREs about the status of the services and its potential impact is the job of monitoring.

AWS Route 53 Application Recovery Controller (ARC) readiness checks validate that the infrastructure required to run the application is configured properly. Readiness (that is, READY or NOT READY) is based on the resources that are in the scope of the readiness check and the set of rules for a resource type. Examine the source code in infrastructure/apps/template/global/route-53-readiness-checks.tf or in the ARC console (after deploying the solution to your AWS account) for a detailed example.

The reference implementation also uses Amazon CloudWatch metrics to monitor key values related to the resilience of the system, such as cross-region replication lag for data stores. If these breach tolerances, an Amazon CloudWatch alarm is raised. These again can be examined in the console after deployment or in the source at infrastructure/apps/common/cloudwatch/alarms.tf

The reference implementation also monitors business-specific metrics emitted by the example applications to ensure the applications are functioning correctly. For example, order turnaround time and matching success rate are continually monitored to ensure that the application is running within service level agreements (SLAs) with business stakeholders.

The consistency of application data is monitored by capturing the number of records in data stores in both regions and presenting them in the dashboard. It monitors the number of orders flowing through the application's components for lags in processing to identify potential bottlenecks before they cause an availability event. The reference implementation runs synthetic transactions through the applications to validate that both the infrastructure and applications are healthy and capable of processing transactions end to end within the acceptable time frame. For specific recommendations on monitoring distributed systems at scale, see the Amazon Builders' Library paper *Instrumenting distributed systems for operational visibility*.

Testing

Regular, thorough testing of the resilience of a system is the only way to verify that it will meet its recovery time and recovery point objectives. The reference implementation demonstrates two distinct types of validation testing.

Failover testing

The reference implementation includes an automated run book implemented as an AWS Systems Manager Document to automate rotation of the active region for both sample applications. In the planned failover scenario, no data loss should occur. This is accomplished by taking a snapshot of the database before and after the rotation and ensuring the two are identical. To test the unplanned scenario, the run book is parameterized to simulate data loss by deleting records. It then validates that the application can recover by invoking a reconciliation and replay process between each pair of producer and consumer microservices within an application and between the matching and settlement applications. These tests are also executed with multiple, distinct sample data loads to ensure that the procedures are successful with different sizes and shapes of application state at the time of failover.

Chaos engineering

Chaos engineering validates a system's ability to be resilient to specific failure scenarios. It is undertaken as a series of experiments. Each experiment introduces a specific failure scenario and monitors how the system responds to such scenario. First, conduct a failure mode analysis exercise to identify the points of

failure, i.e., various things that can go wrong, in the distributed system. Then identify specific remediation steps for each of the failure points and describe the expected behavior when such failover scenario occurs. Each of these scenarios is simulated in an experiment, where the particular failure mode is induced, and the expected behavior is validated.

The reference implementation uses AWS Fault Injection Simulator (FIS) to implement the failure scenario experiments. Each experiment involves identifying a set of target resources, such as the Amazon EC2 instances which make up the ECS cluster hosting the application code. The FIS experiment then takes an action like stopping or terminating the instances to simulate a hardware or other failure. Finally, the experiment verifies that the appropriate actions happen, like restoring capacity to the cluster and raising an Amazon CloudWatch Alarm. The reference implementation does not include exhaustive examples of experiments to test all failure modes but does include multiple example experiments. These leverage both built-in actions supported by FIS and custom actions implemented as run books. For example, experiments update the VPC Endpoint Policy to deny access to specific services required by the application like Amazon Kinesis, Amazon RDS, or Amazon DynamoDB. While not a perfect analog, these evaluate the system's resilience when the health of the AWS service is impaired.

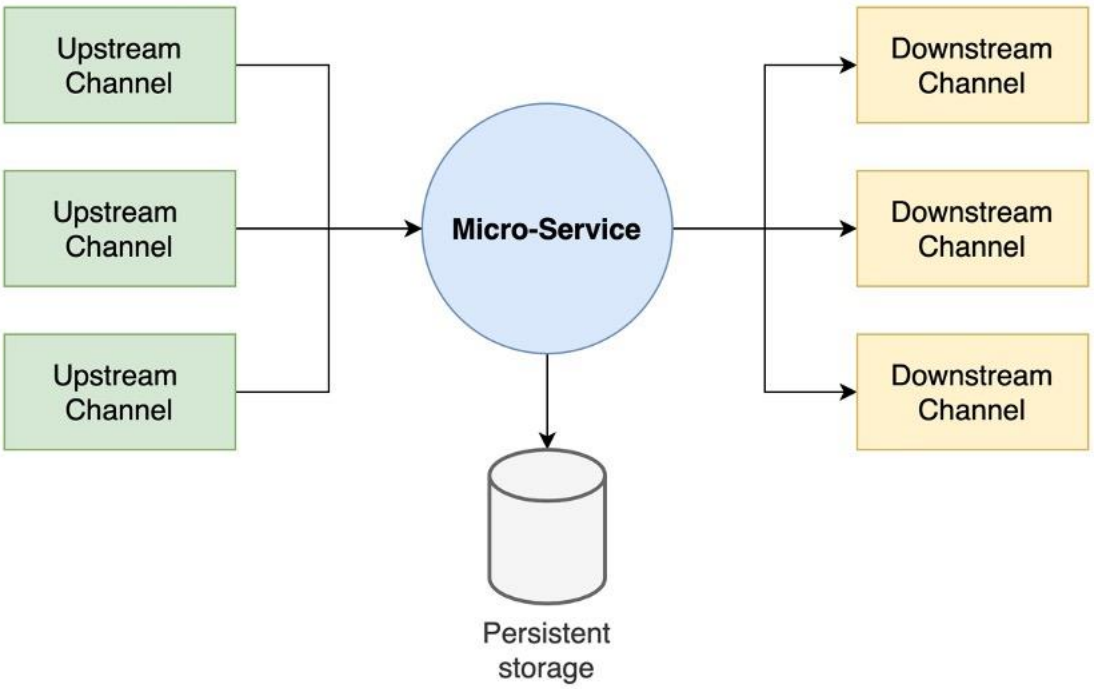
Examine the experiment source in the repository under infrastructure/apps/common/chaos/ or the AWS FIS console (after deploying the reference implementation to your AWS Account) for more details.

Resilient Application Design

This section dives deeper into the application design following the principles defined earlier. Specifically, it describes how a microservices-based approach improves resilience, how the application must respond to specific failure conditions, and why idempotency is important for recovery in a distributed application.

Microservice pattern

An application's resilience is a function of both its own code and configuration – what the application owner controls – and the dependencies it has on infrastructure and other services. It is the application's responsibility to detect and gracefully respond to degraded dependencies. Decomposing the application as a set of loosely coupled, independent microservices is an effective means to achieve this. Such design minimizes the blast radius of a failure to an individual component and will not impact other components and services. The following diagram depicts a high-level view of a microservice with its dependencies and state storage.



Common microservice structure

Examining the diagram reveals several types of failure conditions which the application must detect and handle to be resilient:

- Internal application failure from code defects
- Dependent downstream or upstream resource failure
- Persistent storage failure
- Executing environment failure

The root causes behind each failure type are different, and a developer should consider the specific application use case when implementing a resolution. While defensive coding practices can help resolve issues within the application itself, resolving issues in dependent services is bound by the information and actions that service exposes. Resolutions include trying to fix the failure at runtime, retry processing, or passing the responsibility back to the previous component. In the next sections we will cover some approaches on ways to handle these types of failures.

Exception handling

Proper exception handling is necessary for the application to catch and take appropriate action when a process flow fails to complete successfully. As described earlier, chaos engineering is one of the primary means to validate the exception handling in a distributed application. Exception handling inside the application's codebase is good development practice and there are specific conventions for each

programming language, but special techniques are required for dealing with failures in dependent components outside of the application's control in distributed systems.

Exceptions can be split into two types: Transient and non-transient exceptions.

- **Transient exceptions** are exceptions that could succeed once they are retried, without applying any changes. For example, if an API call times out or the server responds that it is busy.
    - **Retry immediately (Transient):** The first time a failure condition occurs, an immediate retry is often an inexpensive means to resolve the issue. This assumes that the application keeps track of how many times and how recently an error has occurred. However, excessive retries can increase pressure on the dependent system and make the problem worse.
    - **Sleep and try again later (Transient):** A safer way to retry a failed request is to wait for some time before retrying the request. This could allow the downstream channel to clear a processing backlog or resolve some other issue resulting in the failed request. This is often opaque to the client which only receives an error code or the request times out before any response is received. See the Amazon Builders' Library paper *Timeouts, retries, and backoff with jitter* for more details on implementing capped exponential backoff and other retry techniques.
- **Non-transient exceptions** will fail again on retry until the underlying cause of the problem is corrected. For example, if an API responds that the request is improperly formatted, retries with the same data will never succeed.
    - **Skip transaction (Non-Transient):** Rather than continuing to retry processing a request that is failing, it may be best to skip the request after logging it as unprocessed in persistent storage and informing the upstream channel that the request was skipped via specific return values. A compensating process, including human review, can investigate why those requests are failing. See the Transactions Transport State and Checkpoints section below for more details.
    - **Delegate responsibility (Non-Transient):** If the component cannot do the work it was asked to do by retrying or skipping the request, the microservice should indicate this with a critical application error. The monitoring and orchestration mechanisms are then responsible to take corrective action like attempting to restart the microservice or raising an alarm to indicate out-of-region recovery may be necessary.

Applications must be able to differentiate between transient and non-transient failures when catching exceptions in order to determine the appropriate action to take. Proper exception handling reduces the risk that a failed transaction will impact other valid transactions. It also provides the ability to recover from some failures by detecting and retrying failed transactions that can still succeed even though the system may be in a degraded state. Each one of the application's dependent services, such as input services like queues and streams, persistence storage services, or output services like APIs can potentially fail. The application should proactively try to resolve these occurrences when possible and raise an alarm when it is not possible. If it is not possible to handle a failure that is impacting the entire application's health, the application should shut down gracefully as it can no longer process new transactions in its current state.

Transaction state

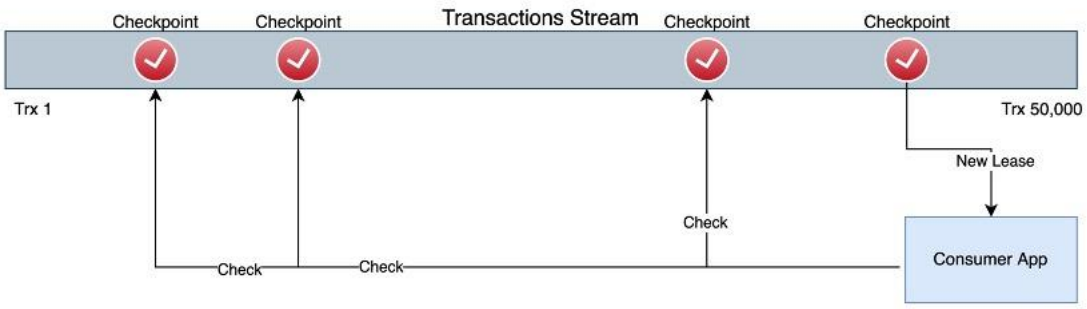
The trade matching and settlement example applications must ensure there is no data loss when processing the securities trading transactions. The state of the overall transaction must be maintained as it is processed by multiple independent microservices as depicted earlier: ingress processing, ingestion, matching, etc. In practice, this means there is a need to persist the transaction details before acknowledging success to an upstream channel. Within each microservice, each transaction is persisted and acknowledged once it is successfully processed. If the transaction did not process successfully, it is automatically assumed that there is a failure, and no acknowledgement is sent to the upstream service. This allows us to compare transaction state across our intermediate states and replay unprocessed transactions.

To make this more concrete, consider the steps of the trade matching process described above. When an investment manager submits the trades and allocations, the matching application, specifically the ingress gateway microservice, should only acknowledge the request as successful once it has confirmation that the request has been persisted to a reliable, resilient data store. As other microservices execute subsequent steps in the process, they must also persist their state, so the overall trade matching transaction can be reconstructed and restarted in the event of a failure. This is explained in more detail in the reconciliation and replay section of this paper.

The reference applications use Amazon MQ for ActiveMQ. This allows microservices to acknowledge successful processing of the message before the message is taken off the processing queue. If a failure occurs after the microservice read the message but before it successfully processed, the message will still be on the queue as unacknowledged when the system is restored. See the Apache ActiveMQ documentation and the Java source for the inbound gateway microservice in the reference implementation repository for more details.

Checkpoints

The prototype also uses Amazon Kinesis to demonstrate an alternative approach to maintain the state of a transaction as it is processed by the microservices. The prototype has implemented checkpoints to safely recover from a failure of the consuming microservice. After a batch of transactions is read from the stream and processed successfully, a checkpoint is created. In the event of recovery after a failure, like the ECS task crashing, the service establishes a new lease and simply reads the last saved checkpoint. Then, it continues processing transactions from that point onwards instead of having to read all transactions from the beginning.



Transaction transport state and checkpoints

Idempotency

What happens when a microservice has successfully processed a transaction message but a failure in either the microservice or the queuing service prevents acknowledgement that the message has been processed? When service is restored, the message will still be on the queue as unacknowledged and the microservice will attempt to process it again. In our example, this could mean a trade order is executed twice. This scenario is potentially worse than a trade not executing at all.

Designing a resilient application requires the application to be able to reprocess transactions because of a failure. But in scenarios like the one described above, it is possible that the same transaction may be processed multiple times. Idempotency is the property that these will yield the same result no matter how many times the same request is submitted.

Common approaches to implementing idempotent microservices are:

- Adding a unique ID per message and configuring the infrastructure to enforce the unique ID constraint to prevent duplicate messages
- Checking whether a record already exists before processing it

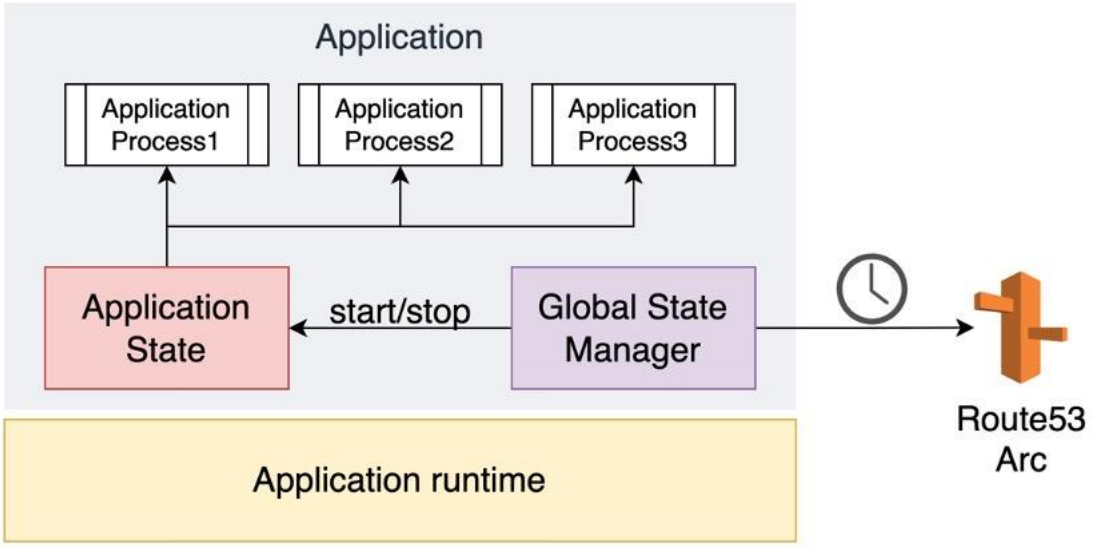
These techniques hold a similar goal of reaching a consistent result, regardless of the number of times we replay a message resulting in a deterministic outcome. For more details, see the AWS Builders' Library paper *Making retries safe with idempotent APIs* and the implementation of the microservices in the reference implementation repository.

Static stability in region

DTCC's regional availability resilience principle requires that an application be architected to continue servicing clients even when some dependent services in the active region are impaired. A pattern called static stability is a key building block to achieve this and describes the ability of the system to remain stable and available even when dependencies are impaired.

Consider the reference implementation's microservices which make up the trade matching process flow. The Amazon ECS clusters which host the microservices' code need a specific amount of capacity to process the expected load. This capacity is distributed among nodes in three AWS Availability Zones (AZ). In the event of an AZ impairment, the nodes in the impaired AZ will fail health checks and the application will stop routing traffic to them until service is restored. All of this occurs without corrective

action, like creating new nodes. For more information, see the AWS Builders' Library paper *Static stability using Availability Zones*.



Implementing application global state manager

Static stability in multiple regions

In addition to being statically stable in a single region, to achieve out-of-region recovery a resilient application must also be statically stable in the second region as well. Effectively, this means that a portion of the compute, database, networking, and other infrastructure resources the application depends on are pre-created in the secondary region so that the failover process is not dependent on creating new resources. Minimizing dependencies increases the success of our failover procedure and decreases the time it takes to recover (RTO).

If an application has redundant services running in two or more independent AWS regions, it also follows that there must be some global service which routes traffic to the "active" region at that time. DNS is the most common service to provide this routing and AWS's managed DNS service, Route 53, is a global service with endpoints in every AWS region. However, the APIs to change DNS records from pointing to IP addresses in one region to those in another are a regional (not global) service. To avoid the situation where a resilience event with DNS prevents the application from failing from one region to the other, a global state manager is needed to allow the application to change routing rules during many failure conditions, including when the primary region is unavailable. Amazon Route 53 Application Recovery Controller (ARC) is a managed global state management service which provides these features.

The implementation can simply be defined by a "start/stop" application interface. This provides our application with the ability of responding to global routing controls in seconds. See the *Global State Manager* section under *Operational Management* in this whitepaper for more information and a detailed diagram.

Additional design recommendations

The following table summarizes specific design principles to achieve stringent RTO and RPO targets for critical systems. Each recommendation is mapped to a resiliency principle category and the risk it mitigates. More details on these recommendations are in the appendix.

Recommendation	Principle category	Risk mitigated
Deploy resources in multiple AWS availability zones (AZ) in both regions	Regional availability Design resilient applications	Service or individual resource availability event in a single AZ affects application availability
Externalize the state of application components to persistent data stores	Regional availability Out-of-region recovery	Exceeding RTO due to data loss or corruption from an application failure
Minimize external dependencies, including AWS Control Plane APIs and Console, during recovery events	Design resilient applications Out-of-region recovery Resilience success	Exceeding RTO due to availability issues in the dependent system
Reserve capacity in both regions	Out-of-region recovery Regional availability	Exceeding RTO due to capacity constraints during a large-scale event
Configure cross-region replication	Design resilient applications Out-of-region recovery	Inability to restore service in second region due to data staleness and exceeding RPO
Automate backup of data stores and test recovery from backup regularly	Out-of-region recovery Regional availability Resilience success	Exceeding RTO due to data corruption
Automate reconciliation of entire system state by replaying in-flight transactions during recovery	Out-of-region recovery Resilience success	Exceeding RPO due to data loss during regional availability event
Design idempotent components	Design resilient applications Regional availability Out-of-region recovery	Data corruption from duplicate transactions during availability events

Recommendation	Principle category	Risk mitigated
Automate DR failover for planned and unplanned regional failover	Out-of-region recovery Resilience success	Exceeding RTO and/or RPO due to human errors while following manual runbook
Communicate with external applications using stable, regionally independent endpoints	Design resilient applications Out-of-region recovery	Exceeding RTO because multiple applications have to be failed over due to an issue with only one of them

Recovery & Rotation

This section covers specific mechanisms to recover and/or rotate a distributed application across regions to a known good state with minimal (optimally zero) data loss. Following a planned or unplanned event, applications require similar procedures to ensure they validate their processing state before accepting new traffic. This entails service health checks, application verification steps, data reconciliation, and a replay process to minimize data loss. In addition, for this effort we chose to leverage one runbook to manage a response to both planned and unplanned events. When preparing for both planned and unplanned events, a single runbook must account for the different steps based on the scenario presented. For example, planned rotations will include graceful application shutdown steps, quiesced data replication cycles, and steps to re-establish data replication in the reverse direction. When executing the same runbook to address an unplanned event, the runbook must detect that the primary region is unavailable and therefore bypass those steps while ensuring it executes reconciliation and replay tasks to account for lost data. The constraints and limitations of delivering an application known good state are discussed below.

Delivering an application design that solves for unplanned (DR) and planned (rotation) events requires careful considerations of the objectives (RPO and RTO). This is true for both in-region events where recovery is required in under 30 minutes and no data loss is acceptable and events that trigger an out-of-region recovery (or rotation) that must be completed in under 2 hours with less than 30 seconds of data loss. Based on these objectives the reference design had to include application specific code, AWS platform services with built-in recovery features where possible, and a fully automated recovery process that sequenced and verified all of the steps required.

Application considerations

To help ensure that the reference architecture design will meet our RTO requirements we chose to implement a hot/warm model with a global traffic manager that redirects transactions to the active region. To meet the RPO requirements we leveraged native AWS services to replicate persistent data and also created app-specific safe stores to help remediate possible data loss following a disruptive event. Application code was developed to perform a reconciliation process that would determine an applications' state following any event, planned or unplanned. Should any data loss be identified an application-specific replay process was developed that pulls data from the persistent storage to close the data loss gap and deliver a known good state.

Data reconciliation

Data reconciliation is the process of ensuring that two sets of records are in agreement. Reconciliation is used to ensure that the transaction output from one persistent store matches the expected next persistence storage. If a failure occurs, resulting in an inability to process the transaction, it may not exist on the next persistence storage. This is the result of the premise that a transaction can only be committed once it is successfully processed.

As a result, differences may be present upon comparing the source and destination of the persistent storage. The data reconciliation process is designed to uncover any potential gaps, and which transactions have failed to process. The reconciliation process can be executed on any two persistence storage repositories. This allows us to logically compare two components in our application.

There is an important distinction between intra-application reconciliation and inter-application reconciliation. Intra-application reconciliation compares any two components within the application, and when a gap is discovered, it can be reasoned regarding the recovery within the application. However, for inter-application reconciliation there is a need to compare two separate applications' persistence storage, and when this is out of sync it will need to be recovered as well as ensure that the downstream application has all the transactions sent by the upstream application.

Data reconciliation needs to operate within a time constraint. Given that the operation could include millions of transactions, the reconciliation process should not be capped at a certain time period when the disruption occurred. This allows effective comparison of a significant smaller amount of data and uses shard based/index for faster performance.

To see a working example of this technique, examine the source for the reconciliation app in the repository.

Data replay

The next part of the application-based recovery implementation is data replay. A replay process was designed to recover missing data resulting from an unplanned event. The application at the recovery site used the results of the reconciliation process to inform the replay of transactions from the safe store. This highlights the need for idempotency so that the transactions can be safely replayed even if some of them have been partially processed. It is also important to understand that when a message is recovered it creates a chain reaction along the microservice processing line. Therefore, before invoking another data reconciliation and replay, it is essential to allow the propagation to finish.

The reference implementation automates the data replay process to run every time after the reconciliation is executed. It is best practice to always execute these steps to help reduce the possibility of data loss after any recovery or rotation event. The inclusion of these automated processes helped the prototype meet the RPO objective by automatically finding and replaying any missing transactions within the RTO. To see a working example, examine the automation document source in the repository under infrastructure/apps/common/rotation/ or the AWS Systems Manager Automation console (after deploying the reference implementation to your AWS Account.)

There are three important considerations that became evident during this effort. First, the reconciliation and replay process should persist all the actions in an auditable fashion in order to provide evidence about missing transactions and the replay of recovered transactions. It also helps with further diagnosis if

required. Second, it should be verified that the entire replay process has been completed before permitting any new transactions into the system. Finally, while the data replay process described was sufficient for this reference implementation, this process may not be sufficient for more complex applications.

Platform considerations

Successful recovery of an application is dependent upon more than just application code that can deliver reconciliation and replay. The prototype design phase highlighted that an application recovery solution can be improved by using the resiliency features of the underlying platform resources. Consider the entire technology stack of both application code and platform resources when designing for maximum resiliency.

Many AWS services offer built-in recovery and failover solutions for in-region and out-of-region events. Utilizing the given recovery functionality can accelerate the recovery time, thus minimizing total RTO/RPO. For example, the trading and settlement sample applications leverage Amazon Aurora Global Database, Amazon Kinesis, and Amazon MQ for ActiveMQ as critical parts of the design:

- **Amazon Aurora Global Database** is designed for globally distributed applications, allowing a single Amazon Aurora database to span multiple AWS Regions. It replicates data with no impact on database performance, enables fast local reads with low latency in each AWS Region, and provides disaster recovery from region-wide outages. Amazon Aurora supports out-of-region recovery through a global-cluster that replicates data between clusters in different AWS Regions, granting customers multi-region resiliency with a built-in failover feature. AWS RDS also offers two mechanisms, including Managed Planned Failover and an unplanned detach and promote pattern, which allows isolating a cluster and enabling it only in a particular AWS Region. Each of these features are useful for a different set of use cases. However, utilizing the Managed Planned Failover holds some additional advantages. For more details, see more about Amazon Aurora Global Disaster Recovery.

  To support the regional availability principle, Amazon Aurora spreads the database storage across multiple AZs, so that an availability event in one AZ will not affect the database service and will automatically recover when the affected AZ is restored to normal operations.

- **Amazon Kinesis** streams offer a built-in checkpoint mechanism, which can be managed by the application directly. Amazon Kinesis holds a checkpoint pointer for each shard, so when a new client connects, it can request Amazon Kinesis to replay all messages from the last checkpoint (trim horizon). As discussed earlier in the Transaction state and checkpoints section, this feature can be utilized so that unless the application moves the checkpoint forward (verified processing), the service will be automatically configured to replay the message. This is an effective technique when an application crashes or environmental availability event occurs during processing and it needs to resume from the last successful processed transaction. This capability only applies to in-region events. This will not apply to loss of region events where recovery will occur in another region.

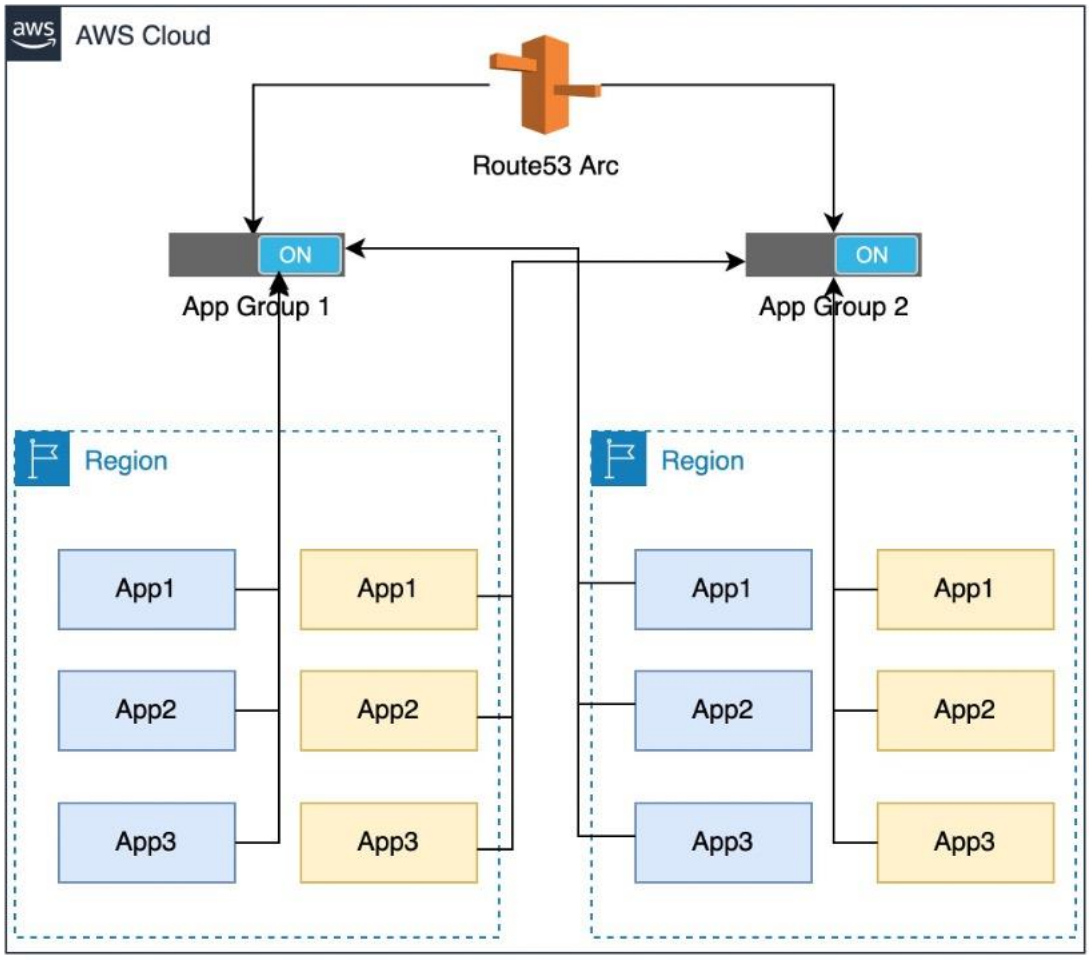
- **Amazon MQ for ActiveMQ** provides several features for recovery and data persistence. Within a single AWS Region, ActiveMQ brokers support Amazon Elastic File System (EFS). Amazon EFS is designed to be highly durable, replicated across multiple AZs to prevent the loss of data resulting from the failure of any single component or an issue that affects the availability of an AZ. In addition, Amazon MQ manages state for each message, so that messages can be replayed when needed. Amazon MQ also supports ActiveMQ's network of brokers feature which can be configured to replicate messages to brokers in a second region.

Operational Management

Having considered the details for designing the infrastructure and applications to support the resilience principles, this section will now consider how site reliability engineers and other operations staff will support these applications in a resilient manner.

Global state management

Managing and operating applications across multiple AWS Regions is not an easy task. It is essential to know when to stop an application in one region and when to resume operating in another; in addition, time is needed for the transition to achieve proper recovery and/or rotation. A highly available control plane such as AWS Route53 ARC provides routing controls and access to change global flags across all the application deployment environments. Nonetheless, it is important to implement the use of these controls both on the DR operation runbook and in the actual application using it.



Controlling applications on multi-region environment

Automating failover with runbooks

Runbooks are our operational orchestration mechanism, consisting of several tasks which are carefully sequenced to achieve a resilient operation such as planned or unplanned disaster recovery (DR).

As mentioned earlier, the reference implementation team built the runbook using an AWS Systems Manager Document. Each step in the runbook invokes a python program which uses the AWS SDK for Python (boto3) to read and update the status of AWS Services. While it is technically possible to automate the decision to failover to the secondary region, DTCC wanted this to be a human decision based on data about the state of the application. Once that go / no-go decision has been made, all the detailed steps are automated in the run book. Operators can verify the status of each step through the dashboard user interface. Examine the automation document source in the repository under infrastructure/apps/common/rotation/ or the custom observability dashboard (after deploying the reference implementation to your AWS Account) for a detailed example of an observable, automated runbook which is pre-provisioned in multiple regions and can be executed in a primary, secondary, or tertiary region.

Planned event – Rotation

A planned failover scenario includes all the tasks necessary to ensure proper transition of the active workload from one region to another with no data loss and with the minimum possible amount of downtime. For this reason, several aspects should be considered on both source and destination regions:

- State of the application
- Messages already in the pipeline (source region)
- State of the resources
- Verify data integrity
- Routing traffic

During a planned failover, there is the benefit of time before executing the transition to ensure any transactions that are currently being processed can be finalized before rotation. In addition, the readiness of the resource and application can be verified to ensure that the data in both regions is consistent and in sync before proceeding with the rotation.

All the relevant tasks were collected into a runbook to execute the rotation. The order of each task was carefully considered, ensuring dependent tasks were timed correctly. Once rotation is completed, operations in the destination region can resume, but are immediately ready to execute rotation back if necessary.

Unplanned event - Disaster Recovery (DR)

A DR scenario is similar to a planned failover but is invoked when reducing the time to restore service (RTO) outweighs trying to recover from problems to avoid data loss (RPO). This decision is based on metrics and the failure types which indicate that waiting for the application to recover in region is unlikely to avoid losing in-flight data. For example, if an AWS service like Amazon Kinesis is experiencing a sustained impairment, waiting may not lead to the streamed data propagating and the unplanned scenario should be invoked. Note that each application must define “sustained” for itself. How long it can wait is a function of its RTO. The same logical order is still followed to ensure the persistent storage is synced and data reconciliation and replay mechanisms are executed in the new region to return the system to a known good state.

The runbook is parameterized to switch between the planned and unplanned scenarios. When the unplanned flag is set, certain steps that wait for data to propagate are skipped and traffic is rerouted as quickly as possible. Since the runbook is deployed on both regions, all of the runbook tasks can be executed from the failover region and allow the regaining of control of the application.

Conclusion

The deliverables of the AWS Global Financial Services team’s engagement with DTCC were:

- A fully functional prototype which adheres to resiliency principles in DTCC’s Application Resiliency Foundation (ARF) program. The RTO measured at less than 30 minutes against a 2-hour target, and RPO was less than 5 seconds against a 30 second target.

- A reconciliation and replay process that was able to recover all of the simulated data loss during a failover process.
- A specific reference implementation code for high availability and disaster recovery in a separate AWS Region for modern applications.
- Open source the code and documentation from that prototype so that all of AWS' customers can install it and use it as a pattern when designing their own resilient systems.

Some of the key takeaways from this experience and this paper are:

1. A highly available orchestration plane is necessary to retain visibility and control of the application, even during an availability event.
2. Comprehensive, repeatable, automated, and verifiable tests of failover procedures are needed to gain the confidence the system is indeed resilient.
3. To prepare for the unknown, it is important to develop capabilities to reconcile data within an application and between applications to a known good state by replaying transactions where necessary.
4. Practice, practice, practice. Since failover is automated and can be conducted at a planned time, do it often so that the decision to invoke a recovery process is an easy one.
5. The collaboration between different engineering teams - development, operational dev-ops, and monitoring - needs to be aligned to orchestrate the final resilience solution.
6. Meeting the required RTO and RPO targets for the application does not depend on a single component or technology resiliency feature, but rather on the composite final solution of all services, technologies, and pipelines. Therefore, it should be evaluated end-to-end with all the intermediate steps required in between.

All cross-region data replication services have limitations and will not by themselves deliver an RPO of zero. The possibility of data loss must be accounted for. AWS provides a multi-regional environment with consistent services, which enables users to orchestrate resilient solution across the globe. DTCC has been a great partner sharing their requirements and challenges, and their unique perspective contributed immensely. This collaboration set high standards that can be used as a guidepost for organizations seeking to achieve better resiliency for their critical workloads on the public cloud.

Contributors

Contributors to this document include:

- Bikash Behera, Enterprise Transformation Architect, Amazon Web Services
- Zahi Ben Shabat, Senior Prototype Architect, Amazon Web Services
- Jack Iu, Principal Solution Architect, Amazon Web Services
- Lee Silverman, Prototyping Engagement Manager, Amazon Web Services
- Jeffrey Quinn, Executive Director of IT Architecture, Depository Trust & Clearing Corporation
- Steven Berube, Director of Application Architecture, Depository Trust & Clearing Corporation
- Abirami Ganeshan, Director of Resiliency Engineering, Depository Trust & Clearing Corporation

- Kenneth Jackson, Senior Manager Solution Architecture, Amazon Web Services
- Deepak Suryanarayanan, Global Manager Multi Region, Amazon Web Services
- Neeraj Kumar, Principal Solution Architect, Amazon Web Services

Document revisions

Date	Description
Month day, 2022	Brief description of revisions.
Month day, YYYY	First publication