AWS Compute Blog

Implementing AWS Well-Architected best practices for Amazon SQS – Part 1

by Pascal Vogel | on 27 JUN 2023 | in Amazon Simple Queue Service (SQS), AWS Well-Architected Framework, Serverless | Permalink | Share

This blog is written by Chetan Makvana, Senior Solutions Architect and Hardik Vasa, Senior Solutions Architect.

Amazon Simple Queue Service (Amazon SQS) is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications. AWS customers have constantly discovered powerful new ways to build more scalable, elastic, and reliable applications using SQS. You can leverage SQS in a variety of use-cases requiring loose coupling and high performance at any level of throughput, while reducing cost by only paying for value and remaining confident that no message is lost. When building applications with Amazon SQS, it is important to follow architectural best practices.

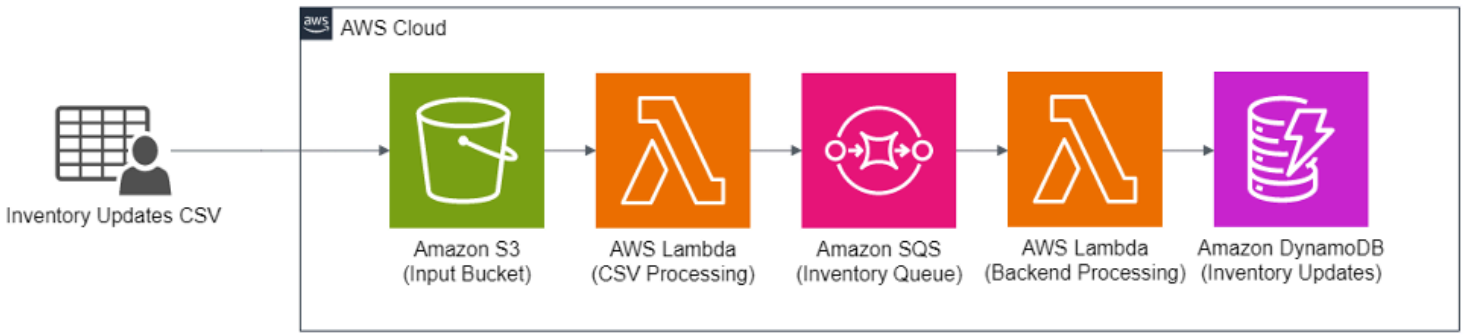
To help you identify and implement these best practices, AWS provides the AWS Well-Architected Framework for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems in the AWS Cloud. Built around six pillars—operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability, AWS Well-Architected provides a consistent approach for customers and partners to evaluate architectures and implement scalable designs.

This three-part blog series covers each pillar of the AWS Well-Architected Framework to implement best practices for SQS. This blog post, part 1 of the series, discusses best practices using the Operational Excellence Pillar of the AWS Well-Architected Framework.

See also the other two parts of the series:

- Implementing AWS Well-Architected Best Practices for Amazon SQS – Part 2: Security and Reliability
- Implementing AWS Well-Architected Best Practices for Amazon SQS – Part 3: Performance Efficiency, Cost Optimization, and Sustainability

Solution overview



This solution architecture shows an example of an inventory management system. The system leverages Amazon Simple Storage Service (Amazon S3), AWS Lambda, Amazon SQS, and Amazon DynamoDB to streamline inventory operations and ensure accurate inventory levels. The system handles frequent updates from multiple sources, such as suppliers, warehouses, and retail stores, which are received as CSV files.

These CSV files are then uploaded to an S3 bucket, consolidating and securing the inventory data for the inventory management system's access. The system uses a Lambda function to read and parse the CSV file, extracting individual inventory update records. The backend Lambda function transforms each inventory update record into a message and sends it to an SQS queue. Another Lambda function continually polls the SQS queue for new messages. Upon receiving a message, it retrieves the inventory update details and updates the inventory levels in DynamoDB accordingly.

This ensures that the inventory quantities for each product are accurate and reflect the latest changes. This way, the inventory management system provides real-time visibility into inventory levels across different locations and suppliers, enabling the company to monitor product availability with precision. Find the example code for this solution in the GitHub repository.

This example is used throughout this blog series to highlight how SQS best practices can be implemented based on the AWS Well Architected Framework.

Operational Excellence Pillar

The Operational Excellence Pillar includes the ability to support development and run workloads effectively, gain insight into their operation, and continuously improve supporting processes and procedures to deliver business value. To achieve operational excellence, the pillar recommends best practices such as defining workload metrics and implementing transaction traceability. This enables organizations to gain valuable insights into their operations, identify potential issues, and optimize services accordingly to improve customer experience. Furthermore, understanding the health of an application is critical to ensuring that it is functioning as expected.

Best practice: Use infrastructure as code to deploy SQS

Infrastructure as Code (IaC) helps you model, provision, and manage your cloud resources. One of the primary advantages of IaC is that it simplifies infrastructure management. With IaC, you can quickly and easily replicate your environment to multiple AWS Regions with a single turnkey solution. This makes it easy to manage your infrastructure, regardless of where your resources are located. Additionally, IaC enables you to create, deploy, and maintain infrastructure in a programmatic, descriptive, and declarative way repeatably. This reduces errors caused by manual processes, such as creating resources in the AWS Management Console. With IaC, you can easily control and track changes in your infrastructure, which makes it easier to maintain and troubleshoot your systems.

For managing SQS resources, you can use different IaC tools like AWS Serverless Application Model (AWS SAM), AWS CloudFormation, or AWS Could Development Kit (AWS CDK). There are also third-party solutions for creating SQS resources, such as the Serverless Framework. AWS CDK is a popular choice because it allows you to provision AWS resources using familiar programming languages such as Python, Java, TypeScript, Go, JavaScript, and C#/.Net.

This blog series showcases the use of AWS CDK with Python to demonstrate best practices for working with SQS. For example, the following AWS CDK code creates a new SQS queue:

Python
```
from aws_cdk import (
    Duration,
    Stack,
    aws_sqs as sqs,
)
from constructs import Construct


class SqsCdBlogStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here

        # example resource
        queue = sqs.Queue(
            self,
            "InventoryUpdatesQueue",
            visibility_timeout=Duration.seconds(300),
        )
```

Best practice: Configure CloudWatch alarms for ApproximateAgeofOldestMessage

It is important to understand Amazon CloudWatch metrics and dimensions for SQS, to have a plan in place to assess its behavior, and to add custom metrics where necessary. Once you have a good understanding of the metrics, it is essential to identify the key metrics that are most relevant to your use case and set up appropriate alerts to monitor them.

One of the key metrics that SQS provides is the `ApproximateAgeOfOldestMessage` metric. By monitoring this metric, you can determine the age of the oldest message in the queue, and take appropriate action to ensure that messages are processed in a timely manner. To set up alerts for the `ApproximateAgeOfOldestMessage` metric, you can use CloudWatch alarms. You configure these alarms to issue alerts when messages remain in the queue for extended periods of time. You can use these alerts to act, for instance by scaling up consumers to process messages more quickly or investigating potential issues with message processing.

In the inventory management example, leveraging the `ApproximateAgeOfOldestMessage` metric provides valuable insights into the health and performance of the SQS queue. By monitoring this metric, you can detect processing delays, optimize performance, and ensure that inventory updates are processed within the desired timeframe. This ensures that your inventory levels remain accurate and up-to-date. The following code creates an alarm which is triggered if the oldest inventory updates request is in the queue for more than 30 seconds.

Python

```
# Create a CloudWatch alarm for ApproximateAgeOfOldestMessage metric
alarm = cloudwatch.Alarm(
    self,
    "OldInventoryUpdatesAlarm",
    alarm_name="OldInventoryUpdatesAlarm",
    metric=queue.metric_approximate_age_of_oldest_message(),
    threshold=600,  # Specify your desired threshold value in seconds
    evaluation_periods=1,
    comparison_operator=cloudwatch.ComparisonOperator.GREATER_THAN_OR_EQUAL_TO_THRESHOl
)
```

Best practice: Add a tracing header while sending a message to the queue to provide distributed tracing capabilities for faster troubleshooting

By implementing distributed tracing, you can gain a clear understanding of the flow of messages in SQS queues, identify any bottlenecks or potential issues, and proactively react to any signals that indicate an unhealthy state. Tracing provides a wider continuous view of an application and helps to follow a user journey or transaction through the application.

AWS X-Ray is an example of a distributed tracing solution that integrates with Amazon SQS to trace messages that are passed through an SQS queue. When using the X-Ray SDK, SQS can propagate tracing headers to maintain trace continuity and enable tracking, analysis, and debugging throughout downstream services. SQS supports tracing headers through the Default HTTP header and the `AWSTraceHeader` System Attribute. `AWSTraceHeader` is available for use even when auto-instrumentation through the X-Ray SDK is not, for example, when building a tracing SDK for a new language. If you are using a Lambda downstream consumer, trace context propagation is automatic.

In the inventory management example, by utilizing distributed tracing with X-Ray for SQS, you can gain deep insights into the performance, behavior, and dependencies of the inventory management system. This visibility enables you to optimize performance, troubleshoot issues more effectively, and ensure the smooth and efficient operation of the system. The following code sets up a CSV processing Lambda function and a backend processing Lambda function with active tracing enabled. The Lambda function automatically receives the X-Ray `TraceId` from SQS.

Python

```
# Create pre-processing Lambda function
csv_processing_to_sqs_function = _lambda.Function(
    self,
    "CSVProcessingToSQSFunction",
    runtime=_lambda.Runtime.PYTHON_3_8,
    code=_lambda.Code.from_asset("sqs_blog/lambda"),
```

```
    handler="CSVProcessingToSQSFunction.lambda_handler",
    role=role,
    tracing=Tracing.ACTIVE,  # Enable active tracing with X-Ray
)

# Create a post-processing Lambda function with the specified role
sqs_to_dynamodb_function = _lambda.Function(
    self,
    "SQSToDynamoDBFunction",
    runtime=_lambda.Runtime.PYTHON_3_8,
    code=_lambda.Code.from_asset("sqs_blog/lambda"),
    handler="SQSToDynamoDBFunction.lambda_handler",
    role=role,
    tracing=Tracing.ACTIVE,  # Enable active tracing with X-Ray
)
```

Conclusion

This blog post explores best practices for SQS with a focus on the Operational Excellence Pillar of the AWS Well-Architected Framework. We explore key considerations for ensuring the smooth operation and optimal performance of applications using SQS. Additionally, we explore the advantages of infrastructure as code in simplifying infrastructure management and showcase how AWS CDK can be used to provision and manage SQS resources.

The next part of this blog post series addresses the Security Pillar and Reliability Pillar of the AWS Well-Architected Framework and explores best practices for SQS.

For more serverless learning resources, visit Serverless Land.
TAGS: contributed, serverless