AWS Compute Blog

Implementing AWS Well-Architected best practices for Amazon SQS – Part 3

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.

This is the third part of a three-part blog post series that demonstrates best practices for Amazon Simple Queue Service (Amazon SQS) using the AWS Well-Architected Framework.

This blog post covers best practices using the Performance Efficiency Pillar, Cost Optimization Pillar, and Sustainability Pillar. The inventory management example introduced in part 1 of the series will continue to serve as an example.

See also the other two parts of the series:

- Implementing AWS Well-Architected Best Practices for Amazon SQS – Part 1: Operation Excellence
- Implementing AWS Well-Architected Best Practices for Amazon SQS – Part 2: Security and Reliability

Performance Efficiency Pillar

The Performance Efficiency Pillar includes the ability to use computing resources efficiently to meet system requirements, and to maintain that efficiency as demand changes and technologies evolve. It recommends best practices to use trade-offs to improve performance, such as learning about design patterns and services and identify how tradeoffs impact customers and efficiency.

By adopting these best practices, you can optimize the performance of SQS by employing appropriate configurations and techniques while considering trade-offs for the specific use case.

Best practice: Use action batching or horizontal scaling or both to increase throughput

For achieving high throughput in SQS, optimizing the performance of your message processing is crucial. You can use two techniques: horizontal scaling and action batching.

When dealing with high message volume, consider horizontally scaling the message producers and consumers by increasing the number of threads per client, by adding more clients, or both. By distributing the load across multiple threads or clients, you can handle a high number of messages concurrently.

Action batching distributes the latency of the batch action over the multiple messages in a batch request, rather than accepting the entire latency for a single message. Because each round trip carries more work, batch requests make more efficient use of threads and connections, improving throughput. You can combine batching with horizontal scaling to provide throughput with fewer threads, connections, and requests than individual message requests.

In the inventory management example that we introduced in part 1, this scaling behavior is managed by AWS for the AWS Lambda function responsible for backend processing. When a Lambda function subscribes to an SQS queue, Lambda polls the queue as it waits for the inventory updates requests to arrive. Lambda consumes messages in batches, starting at five concurrent batches with five functions at a time. If there are more messages in the queue, Lambda adds up to 60 functions per minute, up to 1,000 functions, to consume those messages.

This means that Lambda can scale up to 1,000 concurrent Lambda functions processing messages from the SQS queue. Batching enables the inventory management system to handle a high volume of inventory update messages efficiently. This ensures real-time visibility into inventory levels and enhances the accuracy and responsiveness of inventory management operations.

Best practice: Trade-off between SQS standard and First-In-First-Out (FIFO) queues

SQS supports two types of queues: standard queues and FIFO queues. Understanding the trade-offs between SQS standard and FIFO queues allows you to make an informed choice that aligns with your application's requirements and priorities. While SQS standard queues support a nearly unlimited throughput, it sacrifices strict message ordering and occasionally delivers messages in an order different from the one they were sent in. If maintaining the exact order of events is not critical for your application, utilizing SQS standard queues can provide significant benefits in terms of throughput and scalability.

On the other hand, SQS FIFO queues guarantee message ordering and exactly-once processing. This makes them suitable for applications where maintaining the order of events is crucial, such as financial transactions or event-driven workflows. However, FIFO queues have a lower throughput compared to standard queues. They can handle up to 3,000 transactions per second (TPS) per API method with batching, and 300 TPS without batching. Consider using FIFO queues only when the order of events is important for the application, otherwise use standard queues.

In the inventory management example, since the order of inventory records is not crucial, the potential out-of-order message delivery that can occur with SQS standard queues is unlikely to impact the inventory processing. This allows you to take advantage of the benefits provided by SQS standard queues, including their ability to handle a high number of transactions per second.

Cost Optimization Pillar

The Cost Optimization Pillar includes the ability to run systems to deliver business value at the lowest price. It recommends best practices to build and operate cost-aware workloads that achieve business outcomes while minimizing costs and allowing your organization to maximize its return on investment.

Best practice: Configure cost allocation tags for SQS to organize and identify SQS for cost allocation

A well-defined tagging strategy plays a vital role in establishing accurate chargeback or showback models. By assigning appropriate tags to resources, such as SQS queues, you can precisely allocate costs to different teams or applications. This level of granularity ensures fair and transparent cost allocation, enabling better financial management and accountability.

In the inventory management example, tagging the SQS queue allows for specific cost tracking under the Inventory department, enabling a more accurate assessment of expenses. The following code snippet shows how to tag the SQS queue using AWS Could Development Kit (AWS CDK).

Python
```python
# Create the SQS queue with DLQ setting
queue = sqs.Queue(
    self,
    "InventoryUpdatesQueue",
    visibility_timeout=Duration.seconds(300),
)

Tags.of(queue).add("department", "inventory")
```

Best practice: Use long polling

SQS offers two methods for receiving messages from a queue: short polling and long polling. By default, queues use short polling, where the `ReceiveMessage` request queries a subset of servers to identify available messages. Even if the query found no messages, SQS sends the response right away.

In contrast, long polling queries all servers in the SQS infrastructure to check for available messages. SQS responds only after collecting at least one message, respecting the specified maximum. If no messages are immediately available, the request is held open until a message becomes available or the polling wait time expires. In such cases, an empty response is sent.

Short polling provides immediate responses, making it suitable for applications that require quick feedback or near-real-time processing. On the other hand, long polling is ideal when efficiency is prioritized over immediate feedback. It reduces API calls, minimizes network traffic, and improves resource utilization, leading to cost savings.

In the inventory management example, long polling enhances the efficiency of processing inventory updates. It collects and retrieves available inventory update messages in a batch of 10, reducing the frequency of API requests. This batching approach optimizes resource utilization, minimizes network traffic, and reduces excessive API consumption, resulting in cost savings. You can configure this behavior using batch size and batch window:

Python
```python
# Add the SQS queue as a trigger to the Lambda function
sqs_to_dynamodb_function.add_event_source_mapping(
    "MyQueueTrigger", event_source_arn=queue.queue_arn, batch_size=10
)
```

Best practice: Use batching

Batching messages together allows you to send or retrieve multiple messages in a single API call. This reduces the number of API requests required to process or retrieve messages compared to sending or retrieving messages individually. Since SQS pricing is based on the number of API requests, reducing the number of requests can lead to cost savings.

To send, receive, and delete messages, and to change the message visibility timeout for multiple messages with a single action, use Amazon SQS batch API actions. This also helps with transferring less data, effectively reducing the associated data transfer costs, especially if you have many messages.

In the context of the inventory management example, the CSV processing Lambda function groups 10 inventory records together in each API call, forming a batch. By doing so, the number of API requests is reduced by a factor of 10 compared to sending each record separately. This approach optimizes the utilization of API resources, streamlines message processing, and ultimately contributes to cost efficiency. Following is the code snippet from the CSV processing Lambda function showcasing the use of `SendMessageBatch` to send 10 messages with a single action.

Python
```python
# Parse the CSV records and send them to SQS as batch messages
csv_reader = csv.DictReader(csv_content.splitlines())
message_batch = []
for row in csv_reader:
    # Convert the row to JSON
    json_message = json.dumps(row)

    # Add the message to the batch
    message_batch.append(
        {"Id": str(len(message_batch) + 1), "MessageBody": json_message}
    )

    # Send the batch of messages when it reaches the maximum batch size (10 messages)
    if len(message_batch) == 10:
        sqs_client.send_message_batch(QueueUrl=queue_url, Entries=message_batch)
        message_batch = []
        print("Sent messages in batch")
```

Best practice: Use temporary queues

In case of short-lived, lightweight messaging with synchronous two-way communication, you can use temporary queues. The temporary queue makes it easy to create and delete many temporary messaging destinations without inflating your AWS bill. The key concept behind this is the *virtual queue*. Virtual queues let you multiplex many low-traffic queues onto a single SQS queue. Creating a virtual queue only instantiates a local buffer to hold messages for consumers as they arrive; there is no API call to SQS, and no costs associated with creating a virtual queue.

The inventory management example does not use temporary queues. However, in use cases that involve short-lived, lightweight messaging with synchronous two-way communication, adopting the best practice of using temporary queues and virtual queues can enhance the overall efficiency, reduce costs, and simplify the management of messaging destinations.

Sustainability Pillar

The Sustainability Pillar provides best practices to meet sustainability targets for your AWS workloads. It encompasses considerations related to energy efficiency and resource optimization.

Best practice: Use long polling

Besides its cost optimization benefits explained as part of the Cost Optimization Pillar, long polling also plays a crucial role in improving resource efficiency by reducing API requests, minimizing network traffic, and optimizing resource utilization.

By collecting and retrieving available messages in a batch, long polling reduces the frequency of API requests, resulting in improved resource utilization and minimized network traffic. By reducing excessive API consumption through long polling, you can effectively use resources. It collects and retrieves messages in batches, reducing excessive API consumption and unnecessary network traffic.

By reducing API calls, it optimizes data transfer and infrastructure operations. Additionally, long polling's batching approach optimizes resource allocation, utilizing system resources more efficiently and improving energy efficiency. This enables the inventory management system to handle high message volumes effectively while operating in a cost-efficient and resource-efficient manner.

Conclusion

This blog post explores best practices for SQS using the Performance Efficiency Pillar, Cost Optimization Pillar, and Sustainability Pillar of the AWS Well-Architected Framework. We cover techniques such as batch processing, message batching, and scaling considerations. We also discuss important considerations, such as resource utilization, minimizing resource waste, and reducing cost.

This three-part blog post series covers a wide range of best practices, spanning the Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, and Sustainability Pillars of the AWS Well-Architected Framework. By following these guidelines and leveraging the power of the AWS Well-Architected Framework, you can build robust, secure, and efficient messaging systems using SQS.

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