ADR-001: Use of SQS for Decoupling EventBridge from Lambda Consumer

Status: Accepted

Date: 2025-11-12

Context

The system is required to trigger a validation process (an AWS Lambda function) whenever a new object is created in an S3 bucket. The default, simplest implementation for this is a direct invocation: S3 Event -> EventBridge Rule -> Lambda Function Target.

While functional, this direct invocation pattern presents several resilience and scalability challenges:

1. **Event Loss:** If the target Lambda function fails due to an unhandled exception, throttling, or a downstream service outage, EventBridge will attempt a few retries with exponential backoff. If all retries fail, the event is lost unless a Dead-Letter Queue (DLQ) is configured on the EventBridge rule itself, adding complexity.
2. **Concurrency Spikes ("Thundering Herd"):** A bulk upload of thousands of files to S3 would trigger a corresponding number of concurrent Lambda invocations. This could exhaust the account's concurrency limits, leading to function throttling and failures.
3. **Lack of Buffering:** There is no mechanism to buffer or absorb spikes in traffic. The consumer must handle the load as it comes.

Decision

I have introduced an Amazon SQS (Simple Queue Service) queue as an intermediary between the EventBridge rule and the checksum-validator Lambda function. The workflow will be: S3 Event -> EventBridge Rule -> SQS Queue -> Lambda Function Trigger.

Furthermore, this SQS queue will be configured with its own Dead-Letter Queue (DLQ) to automatically isolate messages that consistently fail processing.

Consequences

Positive:

- **Durability and Reliability:** SQS provides at-least-once message delivery. Once an event is successfully delivered to the SQS queue, it is durably stored and will persist until the Lambda function successfully processes it and deletes it. This virtually eliminates the risk of losing an event due to consumer failure.
- **Resilience and Error Handling:** The SQS-Lambda integration provides an automatic retry mechanism. If the Lambda function fails, the message becomes visible again in the queue for another attempt. The configured DLQ captures "poison pill" messages, preventing a single bad message from blocking the entire validation workflow.
- **Scalability and Concurrency Control:** The SQS queue acts as a shock absorber. It can buffer a massive influx of events from a bulk upload. We can then configure the Lambda's Event Source Mapping to control the batch size (number of messages per invocation) and reserved concurrency, allowing the system to process the backlog at a steady, manageable rate without being overwhelmed.
- **Decoupling:** The event producer (EventBridge) and the consumer (Lambda) are fully decoupled. The producer does not need to know about the consumer's capacity, availability, or implementation details, and vice versa.

Negative:

- **Increased Latency:** Introducing SQS adds a small amount of latency to the workflow (typically milliseconds to a few seconds) compared to a direct invocation. For our backup validation use case, this near-real-time latency is perfectly acceptable.
- **Increased Architectural Complexity:** An additional AWS service is introduced, which must be provisioned, secured (with KMS), and monitored. This adds to the number of components in the system.
- **Slight Cost Increase:** SQS has a cost associated with it, although it is very low and covered by the Free Tier for a significant number of requests. The cost is negligible compared to the resilience benefits.

ADR-002: Use of Customer-Managed Keys (CMK) for Cross-Service Encryption

Status: Accepted

Date: 2025-11-12

Context

Security requirements, enforced by the Checkov security scanner in our CI/CD pipeline, mandate that all data at rest must be encrypted. This applies to our SQS queues and SNS topic. The default and simplest way to enable this is by using AWS-managed keys (e.g., alias/aws/sqs, alias/aws/sns).

However, implementing encryption with AWS-managed keys caused a critical, silent failure in the event flow. The root cause is that the key policies for AWS-managed keys cannot be modified by users. This prevented us from granting the necessary KMS permissions to the AWS service principals that needed to interact with the encrypted resources:

1. **EventBridge (events.amazonaws.com)** could not publish messages because it lacked kms:GenerateDataKey and kms:Decrypt permissions on the SQS key.
2. **Lambda (lambda.amazonaws.com)** could not send failed invocation records to its encrypted DLQ because it lacked kms:CreateGrant permission on the SQS key.
3. The **failure-notifier Lambda's IAM Role** lacked kms:GenerateDataKey permission on the SNS key to publish an encrypted message.

Decision

I have provisioned and use dedicated, **Customer-Managed KMS Keys (CMKs)** for encrypting the SQS queues and the SNS topic.

The key policies for these CMKs will be explicitly defined in our Terraform code. These policies will grant the specific, minimal KMS permissions required by the relevant AWS service principals and IAM roles to perform their functions. A dedicated kms Terraform module was created to manage this.

Consequences

Positive:

- **Enables Functionality:** This decision was necessary to unblock the core functionality of the system. It is the only way to correctly implement end-to-end encryption for this cross-service, event-driven architecture.
- **Granular Security Control:** I have complete control over the key policy. This allows us to enforce the Principle of Least Privilege with precision, specifying exactly which

principals (e.g., events.amazonaws.com) can perform which actions (e.g., kms:GenerateDataKey).

- **Enhanced Auditability:** All API actions (e.g., encryption, decryption) against a CMK are logged in AWS CloudTrail, providing a clear and detailed audit trail for security and compliance.
- **Lifecycle Management:** We control the key's lifecycle, including its rotation schedule, which can be a requirement for certain compliance standards.

Negative:

- **Increased Cost:** Customer-Managed Keys are not free. They incur a monthly cost (e.g., ~$1/month per key) and a per-request cost for API usage. This is a direct cost trade-off for the enhanced security and control.
- **Increased Management Overhead:** We are now fully responsible for the key's policy and lifecycle. A misconfigured key policy could inadvertently lock all principals (including administrators) out of the data, creating a critical availability risk if not managed carefully.
- **Increased Deployment Complexity:** The Terraform configuration became more complex. I had to create a new kms module and update the sqs, sns, and iam modules and their corresponding instantiations to handle the new key dependencies and permissions.