AWS Compute Blog

# Building well-architected serverless applications: Building in resiliency – part 2

by Julian Wood | on 10 AUG 2021 | in Amazon API Gateway, Amazon CloudWatch, Amazon Cognito, Amazon DynamoDB, Amazon EventBridge, Amazon Simple Email Service (SES), Amazon Simple Notification Service (SNS), Amazon Simple Queue Service (SQS), Amazon Simple Storage Service (S3), AWS AppSync, AWS Lambda, AWS Step Functions, AWS Well-Architected Framework, Kinesis Data Streams, Serverless | Permalink | Share

This series of blog posts uses the AWS Well-Architected Tool with the Serverless Lens to help customers build and operate applications using best practices. In each post, I address the serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the introduction post for a table of contents and explanation of the example application.

## Reliability question REL2: How do you build resiliency into your serverless application?

This post continues part 1 of this reliability question. Previously, I cover managing failures using retries, exponential backoff, and jitter. I explain how DLQs can isolate failed messages. I show how to use state machines to orchestrate long running transactions rather than handling these in application code.

## Required practice: Manage duplicate and unwanted events

Duplicate events can occur when a request is retried or multiple consumers process the same message from a queue or stream. A duplicate can also happen when a request is sent twice at different time intervals with the same parameters. Design your applications to process multiple identical requests to have the same effect as making a single request.

Idempotency refers to the capacity of an application or component to identify repeated events and prevent duplicated, inconsistent, or lost data. This means that receiving the same event multiple times does not change the result beyond the first time the event was received. An idempotent application can, for example, handle multiple identical refund operations. The first refund operation is processed. Any further refund requests to the same customer with the same payment reference should not be processes again.

When using AWS Lambda, you can make your function idempotent. The function's code must properly validate input events and identify if the events were processed before. For more information, see "How do I make my Lambda function idempotent?"

When processing streaming data, your application must anticipate and appropriately handle processing individual records multiple times. There are two primary reasons why records may be delivered more than once to your Amazon Kinesis Data Streams application: producer retries and consumer retries. For more information, see "Handling Duplicate Records".

### Generate unique attributes to manage duplicate events at the beginning of the transaction

Create, or use an existing unique identifier at the beginning of a transaction to ensure idempotency. These identifiers are also known as idempotency tokens. A number of Lambda triggers include a unique identifier as part of the event:

- Amazon API Gateway: `requestContext.requestId`
- Kinesis: `Records[].eventID`
- Amazon EventBridge: `id`
- Amazon Simple Notification Service (SNS): `Records[].Sns.MessageId`
- Amazon Simple Queue Service (SQS): `Records[].messageId`

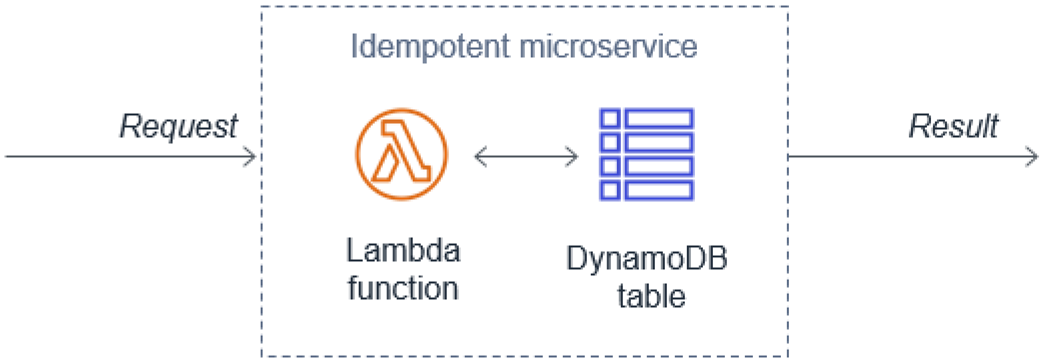
You can also create your own identifiers. These can be business-specific, such as transaction ID, payment ID, or booking ID. You can use an opaque random alphanumeric string, unique correlation identifiers, or the hash of the content.

A Lambda function, for example can use these identifiers to check whether the event has been previously processed.

Depending on the final destination, duplicate events might write to the same record with the same content instead of generating a duplicate entry. This may therefore not require additional safeguards.

## Use an external system to store unique transaction attributes and verify for duplicates

Lambda functions can use Amazon DynamoDB to store and track transactions and idempotency tokens to determine if the transaction has been handled previously. DynamoDB Time to Live (TTL) allows you to define a per-item timestamp to determine when an item is no longer needed. This helps to limit the storage space used. Base the TTL on the event source. For example, the message retention period for SQS.



Using DynamoDB to store idempotent tokens

You can also use DynamoDB conditional writes to ensure a write operation only succeeds if an item attribute meets one of more expected conditions. For example, you can use this to fail a refund operation if a payment reference has already been refunded. This signals to the application that it is a duplicate transaction. The application can then catch this exception and return the same result to the customer as if the refund was processed successfully.

Third-party APIs can also support idempotency directly. For example, Stripe allows you to add an `Idempotency-Key: <key>` header to the request. Stripe saves the resulting status code and body of the first request made for any given idempotency key, regardless of whether it succeeded or failed. Subsequent requests with the same key return the same result.

### Validate events using a pre-defined and agreed upon schema

Implicitly trusting data from clients, external sources, or machines could lead to malformed data being processed. Use a schema to validate your event conforms to what you are expecting. Process the event using the schema within your application code or at the event source when applicable. Events not adhering to your schema should be discarded.

For API Gateway, I cover validating incoming HTTP requests against a schema in "Implementing application workload security – part 1".

Amazon EventBridge rules match event patterns. EventBridge provides schemas for all events that are generated by AWS services. You can create or upload custom schemas or infer schemas directly from events on an event bus. You can also generate code bindings for event schemas.

SNS supports message filtering. This allows a subscriber to receive a subset of the messages sent to the topic using a *filter policy*. For more information, see the documentation.

JSON Schema is a tool for validating the structure of JSON documents. There are a number of implementations available.

## Best practice: Consider scaling patterns at burst rates

Load testing your serverless application allows you to monitor the performance of an application before it is deployed to production. Serverless applications can be simpler to load test, thanks to the automatic scaling built into many of the services. For more information, see "How to design Serverless Applications for massive scale".

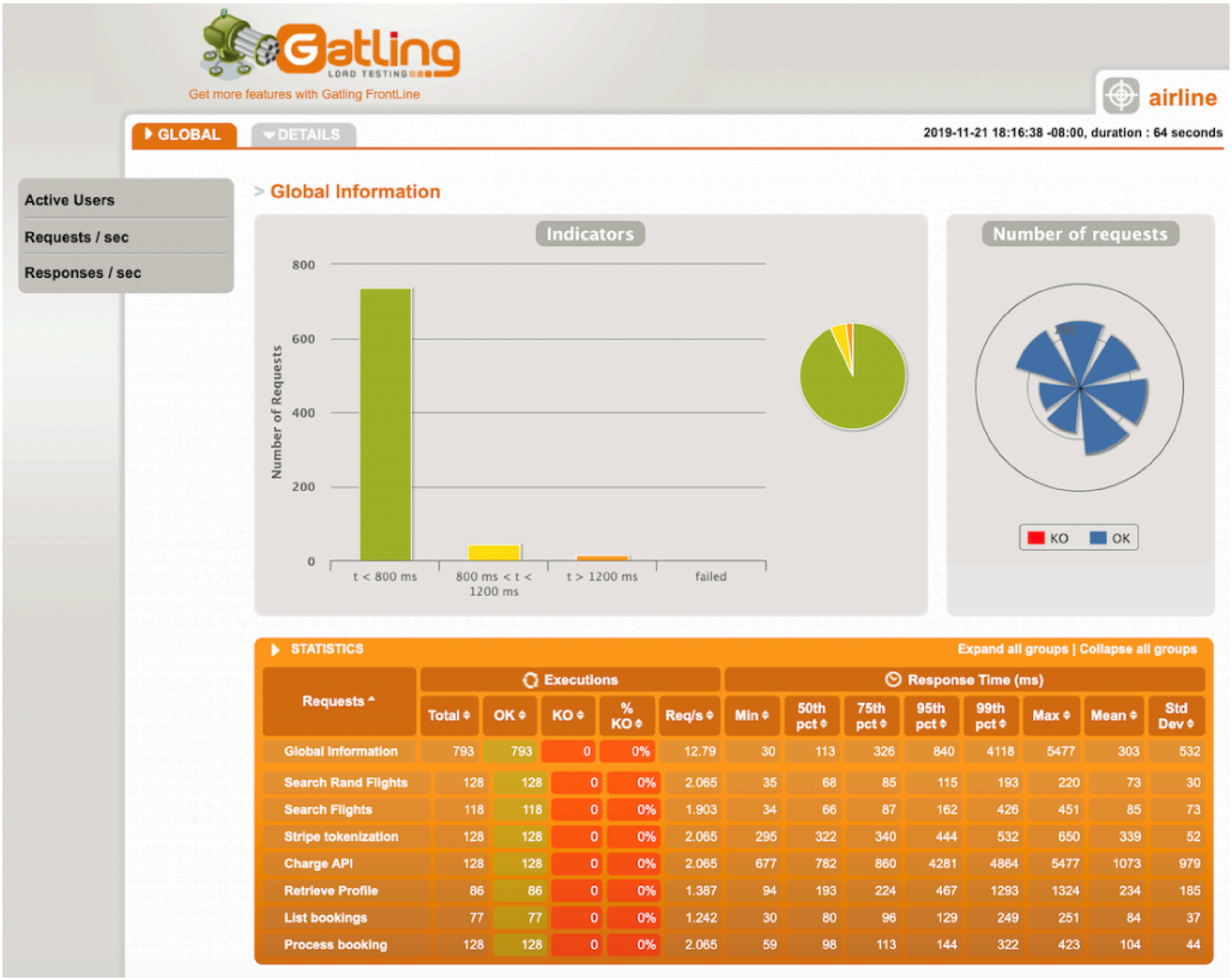
In addition to your baseline performance, consider evaluating how your workload handles initial burst rates. This ensures that your workload can sustain burst rates while scaling to meet possibly unexpected demand.

### Perform load tests using a burst strategy with random intervals of idleness

Perform load tests using a burst of requests for a short period of time. Also introduce burst delays to allow your components to recover from unexpected load. This allows you to future-proof the workload for key events when you do not know peak traffic levels.

There are a number of AWS Marketplace and AWS Partner Network (APN) solutions available for performance testing, including Gatling FrontLine, BlazeMeter, and Apica.

In regulating inbound request rates – part 1, I cover running a performance test suite using Gatling, an open source tool.



STATISTICS

Expand all groups | Collapse all groups

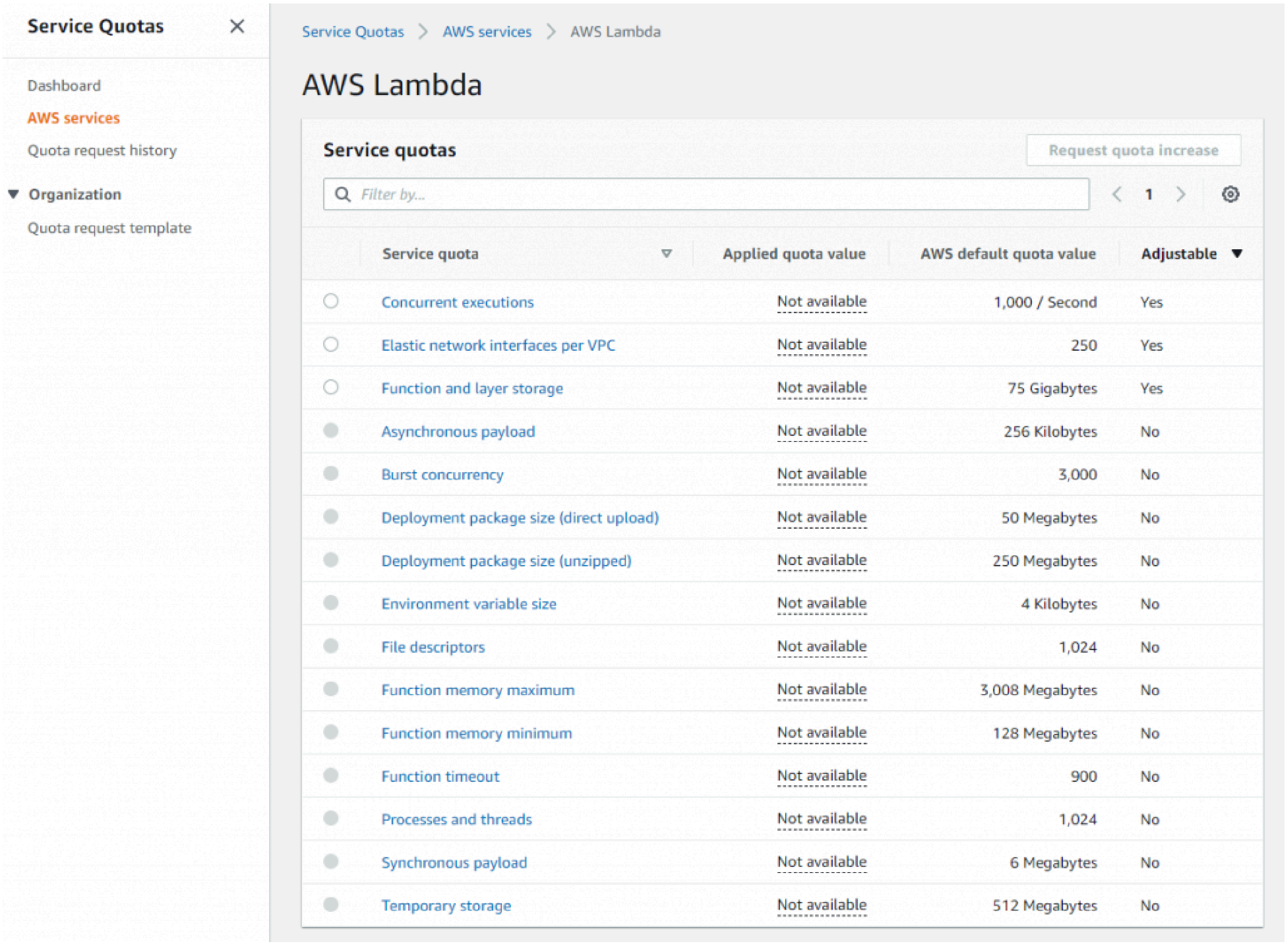
| Requests | Executions | | | | | Response Time (ms) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total | OK | KO | % KO | Req/s | Min | 50th pct | 75th pct | 95th pct | 99th pct | Max | Mean | Std Dev |
| Global Information | 793 | 793 | 0 | 0% | 12.79 | 30 | 113 | 326 | 840 | 4118 | 5477 | 303 | 532 |
| Search Rand Flights | 128 | 128 | 0 | 0% | 2.065 | 35 | 68 | 85 | 115 | 193 | 220 | 73 | 30 |
| Search Flights | 118 | 118 | 0 | 0% | 1.903 | 34 | 66 | 87 | 162 | 426 | 451 | 85 | 73 |
| Stripe tokenization | 128 | 128 | 0 | 0% | 2.065 | 295 | 322 | 340 | 444 | 532 | 650 | 339 | 52 |
| Charge API | 128 | 128 | 0 | 0% | 2.065 | 677 | 782 | 860 | 4281 | 4864 | 5477 | 1073 | 979 |
| Retrieve Profile | 86 | 86 | 0 | 0% | 1.387 | 94 | 193 | 224 | 467 | 1293 | 1324 | 234 | 185 |
| List bookings | 77 | 77 | 0 | 0% | 1.242 | 30 | 80 | 96 | 129 | 249 | 251 | 84 | 37 |
| Process booking | 128 | 128 | 0 | 0% | 2.065 | 59 | 98 | 113 | 144 | 322 | 423 | 104 | 44 |

Gatling performance results

Amazon does have a network stress testing policy that defines which high volume network tests are allowed. Tests that purposefully attempt to overwhelm the target and/or infrastructure are considered distributed denial of service (DDoS) tests and are prohibited. For more information, see “Amazon EC2 Testing Policy”.

## Review service account limits with combined utilization across resources

AWS accounts have default quotas, also referred to as limits, for each AWS service. These are generally Region-specific. You can request increases for some limits while other limits cannot be increased. Service Quotas is an AWS service that helps you manage your limits for many AWS services. Along with looking up the values, you can also request a limit increase from the Service Quotas console.



AWS Lambda

Service quotas

| Service quota | Applied quota value | AWS default quota value | Adjustable |
|---|---|---|---|
| Concurrent executions | Not available | 1,000 / Second | Yes |
| Elastic network interfaces per VPC | Not available | 250 | Yes |
| Function and layer storage | Not available | 75 Gigabytes | Yes |
| Asynchronous payload | Not available | 256 Kilobytes | No |
| Burst concurrency | Not available | 3,000 | No |
| Deployment package size (direct upload) | Not available | 50 Megabytes | No |
| Deployment package size (unzipped) | Not available | 250 Megabytes | No |
| Environment variable size | Not available | 4 Kilobytes | No |
| File descriptors | Not available | 1,024 | No |
| Function memory maximum | Not available | 3,008 Megabytes | No |
| Function memory minimum | Not available | 128 Megabytes | No |
| Function timeout | Not available | 900 | No |
| Processes and threads | Not available | 1,024 | No |
| Synchronous payload | Not available | 6 Megabytes | No |
| Temporary storage | Not available | 512 Megabytes | No |

Service Quotas dashboard

As these limits are shared within an account, review the combined utilization across resources including the following:

- Amazon API Gateway: number of requests per second across all APIs. (link)
- AWS AppSync: throttle rate limits. (link)
- AWS Lambda: function concurrency reservations and pool capacity to allow other functions to scale. (link)
- Amazon CloudFront: requests per second per distribution. (link)
- AWS IoT Core message broker: concurrent requests per second. (link)
- Amazon EventBridge: API requests and target invocations limit. (link)
- Amazon Cognito: API limits. (link)
- Amazon DynamoDB: throughput, indexes, and request rates limits. (link)

## Evaluate key metrics to understand how workloads recover from bursts

There are a number of key Amazon CloudWatch metrics to evaluate and alert on to understand whether your workload recovers from bursts.

- AWS Lambda: *Duration, Errors, Throttling, ConcurrentExecutions, UnreservedConcurrentExecutions.* (link)
- Amazon API Gateway: *Latency, IntegrationLatency, 5xxError, 4xxError*. (link)
- Application Load Balancer: *HTTPCode_ELB_5XX_Count, RejectedConnectionCount, HTTPCode_Target_5XX_Count, UnHealthyHostCount, LambdaInternalError, LambdaUserError*. (link)
- AWS AppSync: *5XX, Latency*. (link)
- Amazon SQS: *ApproximateAgeOfOldestMessage*. (link)
- Amazon Kinesis Data Streams: *ReadProvisionedThroughputExceeded, WriteProvisionedThroughputExceeded, GetRecords.IteratorAgeMilliseconds, PutRecord.Success, PutRecords.Success* (if using Kinesis Producer Library), *GetRecords.Success*. (link)
- Amazon SNS: *NumberOfNotificationsFailed, NumberOfNotificationsFilteredOut-InvalidAttributes*. (link)
- Amazon Simple Email Service (SES): *Rejects, Bounces, Complaints, Rendering Failures*. (link)
- AWS Step Functions: *ExecutionThrottled, ExecutionsFailed, ExecutionsTimedOut*. (link)
- Amazon EventBridge: *FailedInvocations, ThrottledRules*. (link)
- Amazon S3: *5xxErrors, TotalRequestLatency*. (link)
- Amazon DynamoDB: *ReadThrottleEvents, WriteThrottleEvents, SystemErrors, ThrottledRequests, UserErrors*. (link)

## Conclusion

This post continues from part 1 and looks at managing duplicate and unwanted events with idempotency and an event schema. I cover how to consider scaling patterns at burst rates by managing account limits and show relevant metrics to evaluate

Build resiliency into your workloads. Ensure that applications can withstand partial and intermittent failures across components that may only surface in production. In the next post in the series, I cover the performance efficiency pillar from the Well-Architected Serverless Lens.

For more serverless learning resources, visit Serverless Land.

TAGS: serverless, well-architected