AWS Compute Blog

# Building well-architected serverless applications: Building in resiliency – part 1

by Julian Wood | on 03 AUG 2021 | in Amazon DynamoDB, Amazon EventBridge, Amazon Simple Notification Service (SNS), Amazon Simple Queue Service (SQS), AWS CLI, 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?

> Evaluate scaling mechanisms for serverless and non-serverless resources to meet customer demand. Build resiliency into your workload to make your serverless application resilient to withstand partial and intermittent failures across components that may only surface in production.

## Required practice: Manage transaction, partial, and intermittent failures

Whenever one service or system calls another, there is a chance that failures can happen. Services or systems often don't fail as a single unit, but rather suffer partial or transient failures. Applications should be designed to handle component failures as part of the architecture. The system should be designed to detect failure and, ideally, automatically heal itself.

Transaction failures can occur when a component is unavailable or under high load. Partial failures can occur when a percentage of requests succeeds, including during batch processing. Intermittent failures might occur when a request fails for a short period of time due to network or other transient issues.

AWS serverless services, including AWS Lambda, are fault-tolerant and designed to handle failures. If a service invokes a Lambda function and there is a service disruption, Lambda invokes the function in a different Availability Zone.

When you invoke a function directly, you determine the strategy for handling errors. You can retry, send the event to a destination or queue for debugging, or ignore the error. Clients such as the AWS Command Line Interface (CLI) and the AWS SDK retry on client timeouts, throttling errors (429), and other errors that are not caused by a bad request.

When you invoke a function indirectly, you must be aware of the retry behavior of the invoker and any service that the request encounters along the way. For more information, see "Error handling and automatic retries in AWS Lambda". You can configure *Maximum Retry Attempts* and *Maximum Event Age* for asynchronous invocations.

When reading from Amazon Kinesis Data Streams and Amazon DynamoDB Streams, Lambda retries the entire batch of items. Retries continue until the records expire or exceed the maximum age that you configure on the event source mapping. You can also configure the event source mapping to split a failed batch into two batches. Retrying with smaller batches isolates bad records and works around timeout issues.

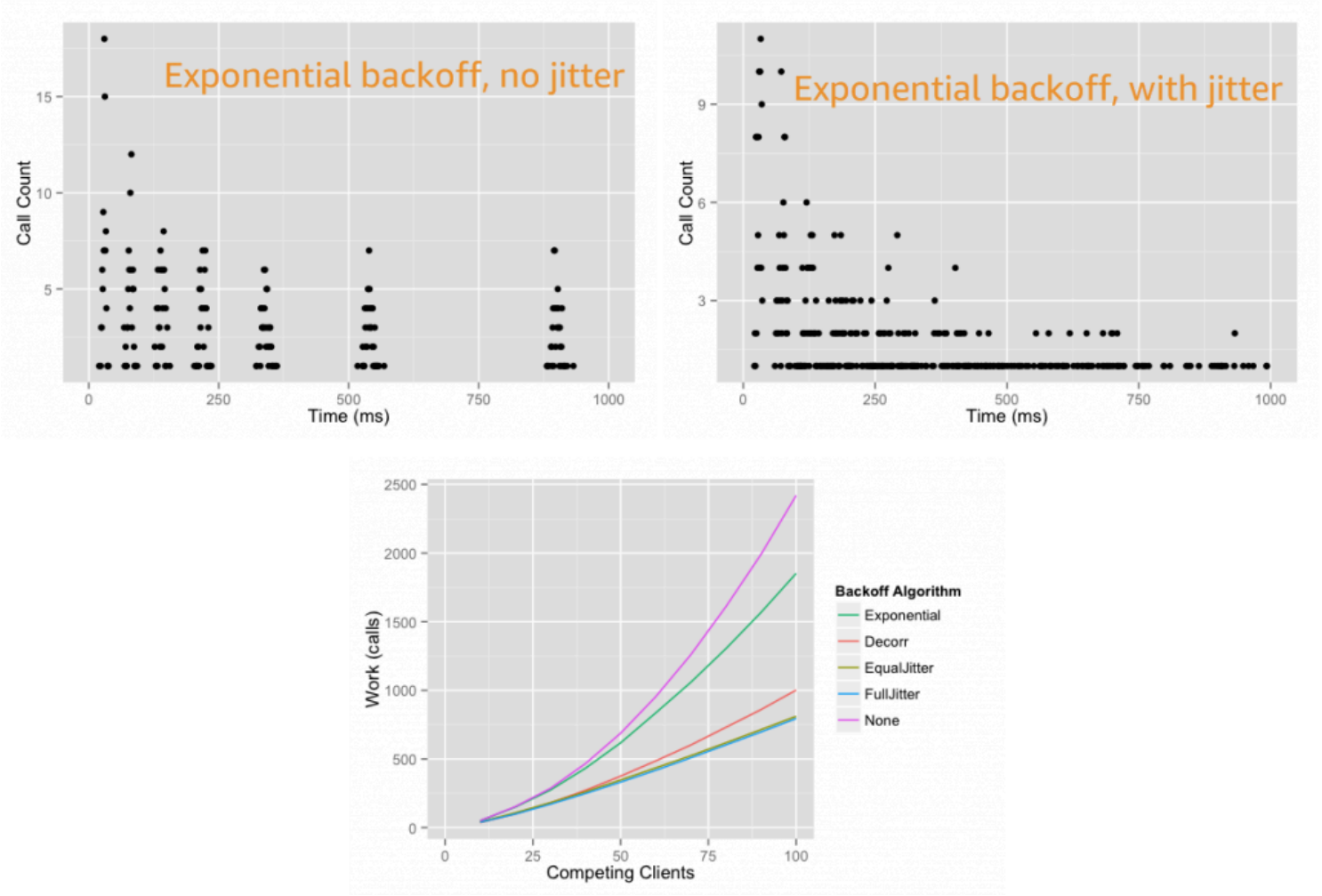
Partial failures can occur in non-atomic operations. `PutRecords` for Kinesis and `BatchWriteItem` for DynamoDB return a successful response if at least one record is ingested successfully. Always inspect the response when using such operations and programmatically deal with partial failures.

## Use exponential backoff with jitter

The simplest technique for dealing with failures in a networked environment is to retry calls until they succeed. This technique increases the reliability of the application and reduces operational costs for the developer.

However, it is not always safe to retry. A retry can further increase the load on the system being called if the system is already failing due to an overload. To avoid this problem, use *backoff*. Instead of retrying immediately and aggressively, the client waits some amount of time between tries. The most common pattern is an *exponential backoff, which* uses exponentially longer wait times between retries. This is typically capped to a maximum delay and number of retries.

If all backoff retries are still happening at the same time, this can still overload a system or cause contention. To avoid this problem, use jitter. Jitter adds some amount of randomness to the backoff to spread the retries around in time. This can help prevent large bursts by spreading out the rate when clients connect. For more information see the Amazon Builders' Library article "Timeouts, retries, and backoff with jitter" and AWS Architecture blog post "Exponential Backoff And Jitter".




Exponential backoff and jitter

When your application responds to callers in fail-fast scenarios and when performance is degraded, inform the caller via headers or metadata when they can retry.

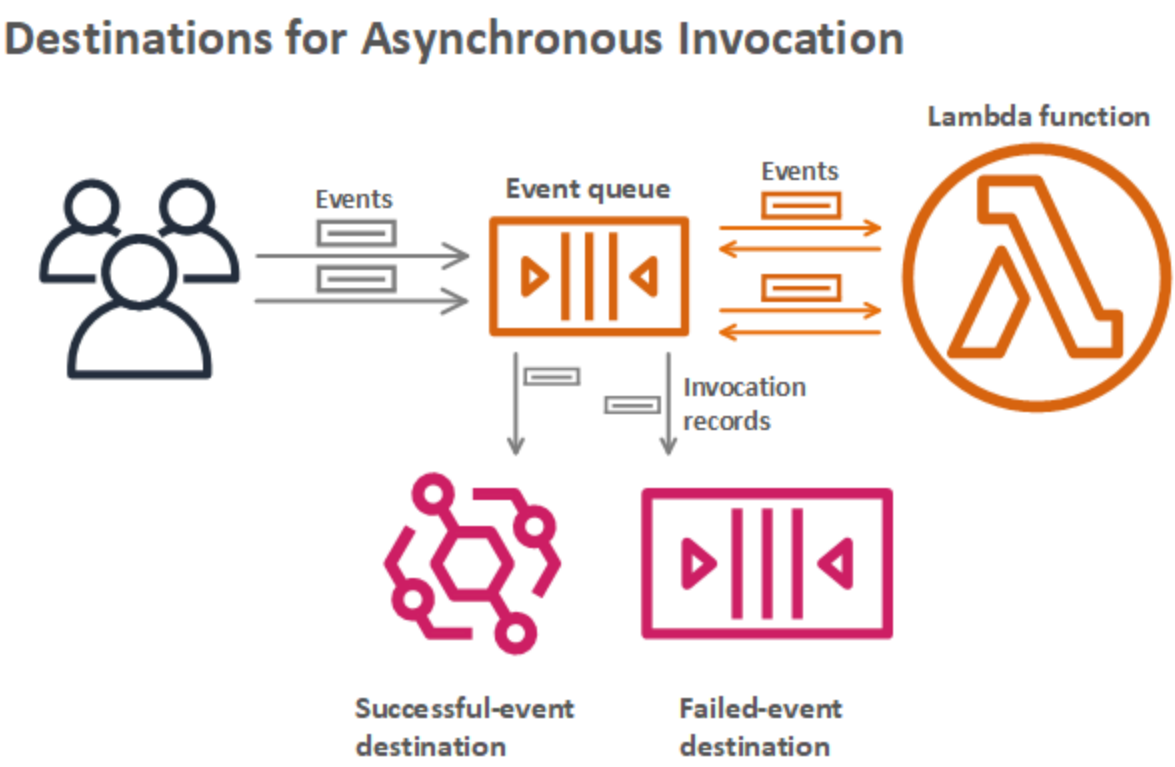
Each AWS SDK implements automatic retry logic including exponential backoff. For downstream calls, you can adjust AWS and third-party SDK retries, backoffs, TCP, and HTTP timeouts. This helps you decide when to stop retrying. For more information, see the documentation and troubleshooting steps for Lambda and the AWS SDK.

## Use a dead-letter queue mechanism to retain, investigate and retry failed transactions

There are a number of ways to handle message failures including destinations and dead-letter queues.

You can configure Lambda to send records of asynchronous invocations to another destination service. These include Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS), Lambda, and Amazon EventBridge. You can configure separate destinations for events that fail processing and events that are successfully processed. The invocation record contains details about the event, the response, and the reason that the record was sent.

The following example shows a function that sends a record of a successful invocation to an EventBridge event bus. When an event fails all processing attempts, Lambda sends an invocation record to an SQS queue. It includes the function's response in the invocation record.




AWS Lambda destinations for asynchronous invocation

SNS, SQS, Lambda, and EventBridge support dead-letter queues (DLQs). DLQs make your applications more resilient and durable by storing messages or events that can't be processed correctly into a dedicated SQS queue. This helps you debug your application by isolating the problematic messages to determine why their processing failed. One you have resolved the issue, re-process the failed message. For more information, see "When should I use a dead-letter queue?" There is an example serverless application to redrive the messages from an SQS DLQ back to its source SQS queue.

For Lambda, DLQs provide an alternative to a failure destination. Lambda destinations is preferable for asynchronous invocations.

## Good practice: Orchestrate long-running transactions

Long-running transactions can be processed by one or multiple components. Consider implementing the saga pattern using state machines for these types of transactions.

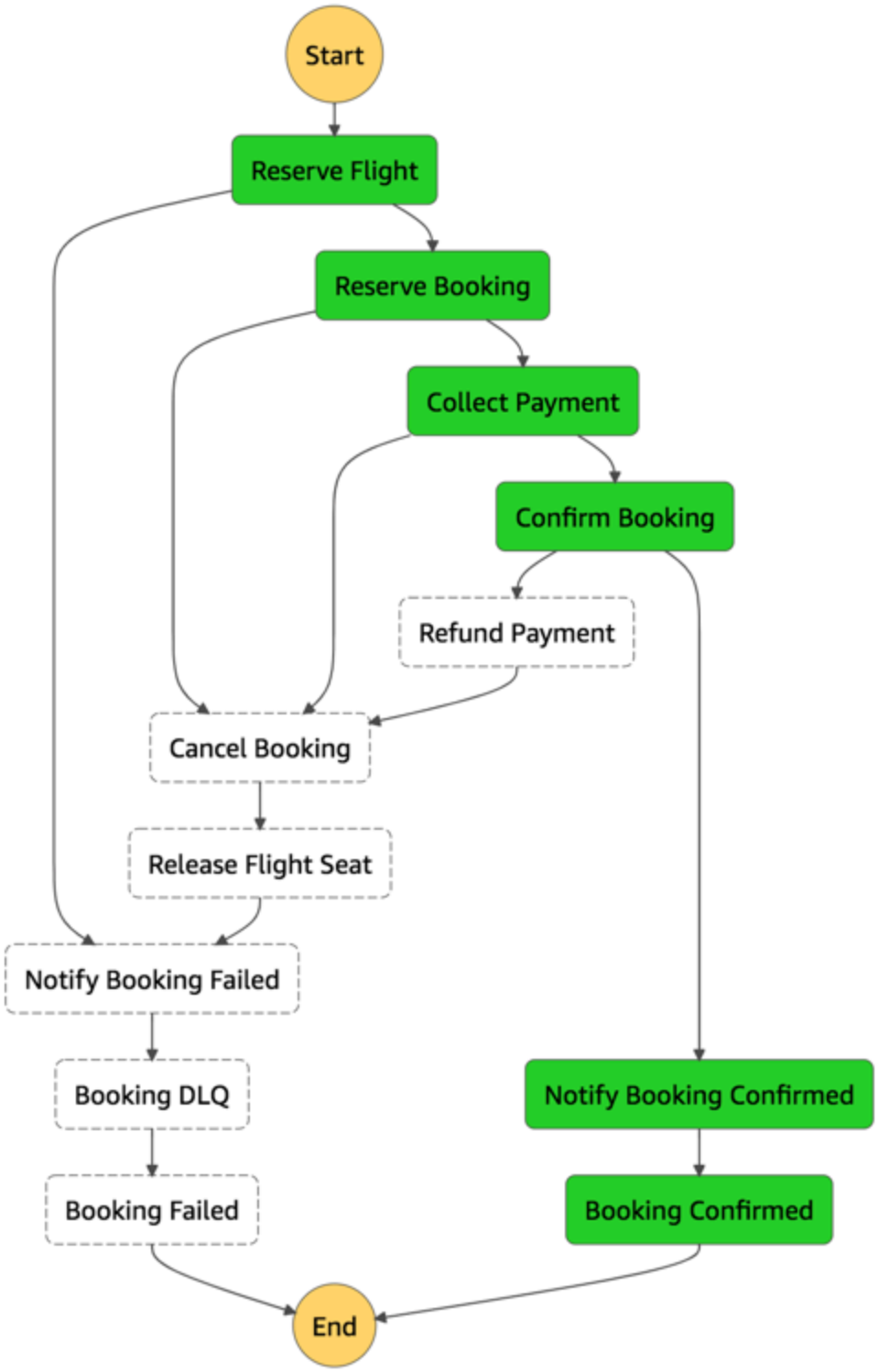
The saga pattern coordinates transactions between multiple microservices as part of a state machine. Each service that performs a transaction publishes an event to trigger the next transaction in the saga. This continues until the transaction chain is complete. If a transaction fails, saga orchestrates a series of compensating transactions that undo the changes that were made by the preceding transactions.

This is preferable to handling complex or long-running transactions within application code. State machines prevent cascading failures and avoid tightly coupling components with orchestrating logic and business logic.

## Use a state machine to visualize distributed transactions, and to separate business logic from orchestration logic.

AWS Step Functions lets you coordinate multiple AWS services into serverless workflows via state machines. Within Step Functions, you can set separate retries, backoff rates, max attempts, intervals, and timeouts. These are set for every step of your state machine using a declarative language.

In the serverless airline example used in this series, Step Functions is used to orchestrate the Booking microservice. The ProcessBooking state machine handles all the necessary steps to create bookings, including payment.



Booking service Step Functions state machine

The state machine uses a combination of service integrations using DynamoDB, SQS, and Lambda functions to coordinate transactions and handle failures.

For example, the Reserve Booking task invokes a Lambda function. The task has retry and error handling configured as part of the task definition.

JSON

```
"Reserve Booking": {
    "Type": "Task",
    "Resource": "${ReserveBooking.Arn}",
    "TimeoutSeconds": 5,
    "Retry": [
        {
            "ErrorEquals": [
                "BookingReservationException"
            ],
            "IntervalSeconds": 1,
            "BackoffRate": 2,
            "MaxAttempts": 2
        }
    ],
    "Catch": [
        {
            "ErrorEquals": [
                "States.ALL"
            ],
```

Step Functions supports direct service integrations, including DynamoDB. The Reserve Flight task directly updates the `flightTable` without requiring a Lambda function.

JSON

```
"Reserve Flight": {
    "Type": "Task",
    "Resource": "arn:aws:states:::dynamodb:updateItem",
    "Parameters": {
        "TableName.$": "$.flightTable",
        "Key": {
            "id": {
                "S.$": "$.outboundFlightId"
            }
        },
        "UpdateExpression": "SET seatCapacity = seatCapacity - :dec",
        "ExpressionAttributeValues": {
            ":dec": {
```

```
            "N": "1"
        },
        ":noSeat": {
            "N": "0"
        }
```

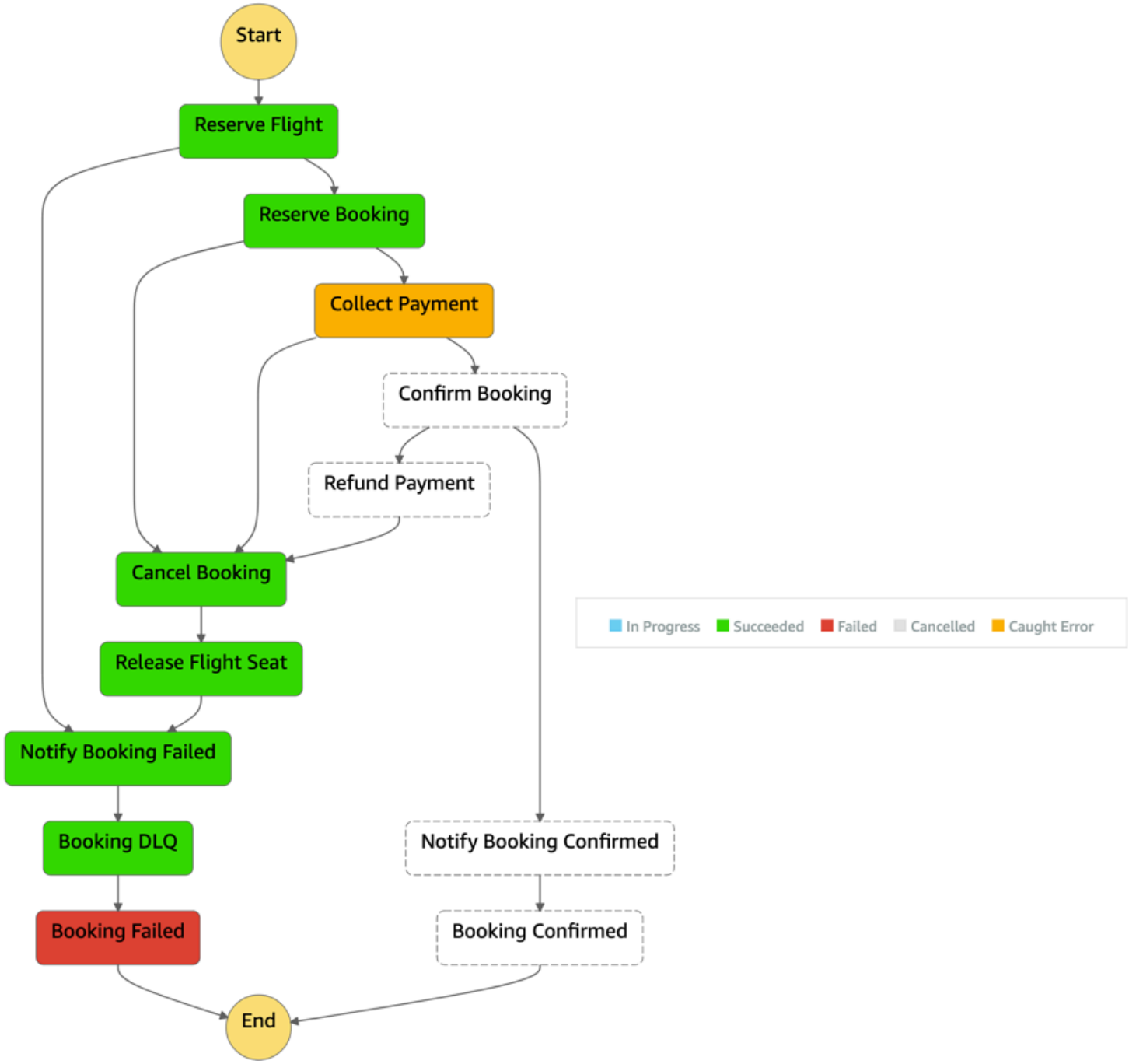
By default, when a state reports an error, Step Functions causes the execution to fail entirely.

## Utilize dead-letter queues in response to failed state machine executions

Any state within the Step Functions workflow can encounter runtime errors. These include state machine definiti issues, task failures such as Lambda function exceptions, or transient issues such as network connectivity issues. more information, see "Error handling in Step Functions".

Use the Step Functions service integration with SQS to send failed transactions to a DLQ as the final step. This a a higher level of durability within your state machines.

For example, the airline Notify Failed Booking final task catches failed states from four previous steps. It sends th results to the Booking DLQ.




Booking service Step Functions DLQ

The message includes the output of the previous failed states for further troubleshooting.

JSON

```
"Booking DLQ": {
    "Type": "Task",
    "Resource": "arn:aws:states:::sqs:sendMessage",
    "Parameters": {
        "QueueUrl": "${BookingsDLQ}",
        "MessageBody.$": "$"
    },
    "ResultPath": "$.deadLetterQueue",
```

```
    "Next": "Booking Failed"
},
```

The Step Functions documentation has more information on calling SQS.

## Conclusion

Build resiliency into your workloads. This makes sure that your application can withstand partial and intermittent failures across components that may only surface in production.

In this post, 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.

This well-architected question continues in part 2 where I look 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.

For more serverless learning resources, visit Serverless Land.
TAGS: serverless, well-architected