

How to fail with Serverless

Jeremy Daly

CTO, AlertMe.news

 @jeremy_daly

Jeremy Daly



Jeremy Daly

- CTO at AlertMe.news



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups
- Started working with AWS in 2009 and started using Lambda in 2015



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups
- Started working with AWS in 2009 and started using Lambda in 2015
- Blogger (jeremydaly.com), OSS contributor, speaker



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups
- Started working with AWS in 2009 and started using Lambda in 2015
- Blogger (jeremydaly.com), OSS contributor, speaker
- Publish the **Off-by-none** serverless newsletter



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups
- Started working with AWS in 2009 and started using Lambda in 2015
- Blogger (jeremydaly.com), OSS contributor, speaker
- Publish the **Off-by-none** serverless newsletter
- Host of the **Serverless Chats** podcast



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups
- Started working with AWS in 2009 and started using Lambda in 2015
- Blogger (jeremydaly.com), OSS contributor, speaker
- Publish the **Off-by-none** serverless newsletter
- Host of the **Serverless Chats** podcast



Jeremy Daly

- CTO at AlertMe.news
- Consult with companies building in the cloud
- 20+ year veteran of technology startups
- Started working with AWS in 2009 and started using Lambda in 2015
- Blogger (jeremydaly.com), OSS contributor, speaker
- Publish the **Off-by-none** serverless newsletter
- Host of the **Serverless Chats** podcast



Agenda

Agenda

- Distributed systems and **serverless**

Agenda

- Distributed systems and **serverless**
- Writing code **FOR** the cloud

Agenda

- Distributed systems and **serverless**
- Writing code **FOR** the cloud
- **Failure modes** in the cloud

Agenda

- Distributed systems and **serverless**
- Writing code **FOR** the cloud
- **Failure modes** in the cloud
- Serverless **patterns** to deal with failure

Distributed Systems are...

Distributed Systems are...

Systems whose components are located on **different networked computers**, which communicate and coordinate their actions by **passing messages** to one another. ~ *Wikipedia*

Distributed Systems are...

Systems whose components are located on **different networked computers**, which communicate and coordinate their actions by **passing messages** to one another. ~ *Wikipedia*

They're also **really hard!**



Werner Vogels
CTO, Amazon.com

EVERYTHING FAILS ALL THE TIME


Werner Vogels
CTO, Amazon.com



#qa-jeremy-daly

ed the Turing tes

FAILOVER CONF.

 @jeremy_daly

Serverless applications are...

Serverless applications are...

Distributed systems on **steroids!** 🦵 🦵 🦵

Serverless applications are...

Distributed systems on **steroids!** 

- Smaller, more **distributed compute** units

Serverless applications are...

Distributed systems on **steroids!** 🦵 🦵 🦵

- Smaller, more **distributed compute** units
- **Stateless**, requiring network access to state

Serverless applications are...

Distributed systems on **steroids!** 🦵 🦵 🦵

- Smaller, more **distributed compute** units
- **Stateless**, requiring network access to state
- **Uncoordinated**, requires buses, queues, pub/sub, state machines

Serverless applications are...

Distributed systems on **steroids!** 🦵 🦵 🦵

- Smaller, more **distributed compute** units
- **Stateless**, requiring network access to state
- **Uncoordinated**, requires buses, queues, pub/sub, state machines
- Heavily reliant on other **networked cloud services**

What does it mean to be **Serverless**?

What does it mean to be **Serverless**?

- No server management

What does it mean to be **Serverless**?

- No server management
- Flexible scaling

What does it mean to be **Serverless**?

- No server management
- Flexible scaling
- Pay for value / never pay for idle

What does it mean to be **Serverless**?

- No server management
- Flexible scaling
- Pay for value / never pay for idle
- Automated high availability

What does it mean to be **Serverless**?

- No server management
- Flexible scaling
- Pay for value / never pay for idle
- Automated high availability
- **LOTS** of configuration and knowledge of cloud services

What does it mean to be **Serverless**?

- No server management
- Flexible scaling
- Pay for value / never pay for idle
- Automated high availability
- **LOTS** of configuration and knowledge of cloud services
- Highly **event-driven**

Lots of services to communicate with!

Lots of services to communicate with!

"Serverless"



Lambda



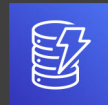
Cognito



Kinesis



S3



DynamoDB



SQS



SNS



API Gateway



EventBridge



AppSync



IoT



Comprehend

Lots of services to communicate with!

"Serverless"



Lambda



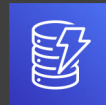
Cognito



Kinesis



S3



DynamoDB



SQS



SNS



API Gateway



EventBridge



AppSync

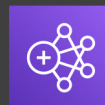


IoT

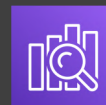


Comprehend

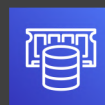
Managed



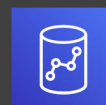
EMR



Amazon ES



ElastiCache



Redshift



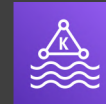
RDS



Fargate



DocumentDB
(MongoDB)



Managed Streaming
for Kafka

Lots of services to communicate with!

"Serverless"



Lambda



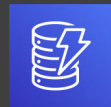
Cognito



Kinesis



S3



DynamoDB



SQS



SNS



API Gateway



EventBridge



AppSync



IoT

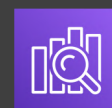


Comprehend

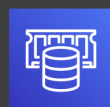
Managed



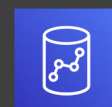
EMR



Amazon ES



ElastiCache



Redshift



RDS



Fargate



DocumentDB
(MongoDB)



Managed Streaming
for Kafka

Non-Serverless



Anything "on EC2"



cassandra



docker



kafka



Microsoft
SQL Server



mongoDB

Reliability or High Availability is...

Reliability or High Availability is...

A characteristic of a system, which aims to ensure an agreed level of **operational performance**, usually **uptime**, for a higher than normal period.

~ Wikipedia

Resiliency is...

Resiliency is...

The ability of a software solution to **absorb the impact** of a problem in one or more parts of a system, while continuing to provide an **acceptable service level** to the business. ~ *IBM*

Resiliency is...

The ability of a software solution to **absorb the impact** of a problem in one or more parts of a system, while continuing to provide an **acceptable service level** to the business. ~ *IBM*

IT'S NOT ABOUT PREVENTING FAILURE

Resiliency is...

The ability of a software solution to **absorb the impact** of a problem in one or more parts of a system, while continuing to provide an **acceptable service level** to the business. ~ *IBM*

IT'S NOT ABOUT PREVENTING FAILURE

IT'S UNDERSTANDING HOW TO GRACEFULLY DEAL WITH IT

Writing code FOR the Cloud

Using **Lambda** for our business logic



Using **Lambda** for our business logic

- **Ephemeral** compute service



Using **Lambda** for our business logic

- **Ephemeral** compute service
- Runs your code in **response to events**



Using **Lambda** for our business logic

- **Ephemeral** compute service
- Runs your code in **response to events**
- Automatically **manages** the runtime, compute, and scaling



Using **Lambda** for our business logic

- **Ephemeral** compute service
- Runs your code in **response to events**
- Automatically **manages** the runtime, compute, and scaling
- **Single concurrency** model



Using **Lambda** for our business logic

- **Ephemeral** compute service
- Runs your code in **response to events**
- Automatically **manages** the runtime, compute, and scaling
- **Single concurrency** model
- No **sticky-sessions** or guaranteed **lifespan**



Traditional Error Handling & Retries

Traditional Error Handling & Retries

```
try {  
    // Do something important  
} catch (err) {  
    // Do some error handling  
    // Do some logging  
    // Maybe retry the operation  
}
```

Traditional Error Handling & Retries

```
try {  
    // Do something important  
} catch (err) {  
    // Do some error handling  
    // Do some logging  
    // Maybe retry the operation  
}
```

What happens to the
original event?

Traditional Error Handling & Retries

```
try {  
    // Do something important  
} catch (err) {  
    // Do some error handling  
    // Do some logging  
    // Maybe retry the operation  
}
```

What happens to the **original** event?

What happens if there is a **network issue**?

Traditional Error Handling & Retries

```
try {  
    // Do something important  
} catch (err) {  
    // Do some error handling  
    // Do some logging  
    // Maybe retry the operation  
}
```

What happens to the **original** event?

What happens if there is a **network issue**?

What happens if the function container **crashes**?

Traditional Error Handling & Retries

```
try {  
    // Do something important  
} catch (err) {  
    // Do some error handling  
    // Do some logging  
    // Maybe retry the operation  
}
```

What happens to the **original** event?

What happens if there is a **network issue**?

What happens if the function container **crashes**?

What happens if the function **never runs**?

Traditional Error Handling & Retries

```
try {  
    // Do something important  
} catch (err) {  
    // Do some error handling  
    // Do some logging  
    // Maybe retry the operation  
}
```

Losing events is **very bad!**

What happens to the **original** event?

What happens if there is a **network issue**?

What happens if the function container **crashes**?

What happens if the function **never runs**?

Function **Error** Types



Function **Error** Types

- Unhandled Exception **!!**




Function **Error** Types

- Unhandled Exception **!!**
- Function Timeouts 

Function **Error** Types

- Unhandled Exception **!!**
- Function Timeouts 
- Out-of-Memory Errors 

Function **Error** Types

- Unhandled Exception **!!**
- Function Timeouts 
- Out-of-Memory Errors 
- Throttling Errors 

The **cloud** is better than you...

The **cloud** is better than you...

...at error handling

The **cloud** is better than you...

...at error handling

...at retrying failures

The **cloud** is better than you...

...at error handling

...at retrying failures

...at understanding network failures

The **cloud** is better than you...

...at error handling

...at retrying failures

...at understanding network failures

...at mapping the network topology

The **cloud** is better than you...

...at error handling

...at retrying failures

...at understanding network failures

...at mapping the network topology

...at handling failover and redundancy

The **cloud** is better than you...

...at error handling

...at retrying failures

...at understanding network failures

...at mapping the network topology

...at handling failover and redundancy

So why not let the **cloud** do those things **for you?** 🙄

Fail up the stack

Fail up the stack

- Don't swallow errors with try/catch – **fail the function**

Fail up the stack

- Don't swallow errors with try/catch – **fail the function** (sometimes 🤔)

Fail up the stack

- Don't swallow errors with try/catch – **fail the function** (sometimes 🤔)
- **Return errors** directly to the invoking service

Fail up the stack

- Don't swallow errors with try/catch – **fail the function** (sometimes 🤔)
- **Return errors** directly to the invoking service
- Configure built-in **retry mechanisms** to reprocess events

Fail up the stack

- Don't swallow errors with try/catch – **fail the function** (sometimes 🤔)
- **Return errors** directly to the invoking service
- Configure built-in **retry mechanisms** to reprocess events
- Utilize **dead letter queues** to capture failed events

Types of **Lambda** Functions

Types of **Lambda** Functions

- The **Lambdalith**

Types of **Lambda** Functions

- The **Lambdalith** 😬

Types of **Lambda** Functions

- The **Lambdalith** 😬
- The **Fat Lambda**

Types of **Lambda** Functions

- The **Lambdalith** 🤪
- The **Fat Lambda** 🤔

Types of **Lambda** Functions

- The **Lambdalith** 🤪
- The **Fat Lambda** 🤔
- The **Single-Purpose** Function

Types of **Lambda** Functions

- The **Lambdalith** 🤪
- The **Fat Lambda** 🤔
- The **Single-Purpose** Function 👍

The Mighty **Lambdalith**

The Mighty **Lambdalith**



The Mighty **Lambdalith**

- The entire application is in **one** Lambda function



The Mighty **Lambdalith**

- The entire application is in **one** Lambda function
- Often times these are “**lift and shift**” Express.js or Flask apps



The Mighty **Lambdalith**

- The entire application is in **one** Lambda function
- Often times these are “**lift and shift**” Express.js or Flask apps
- Events are **synchronous** via API Gateway or ALB



The Mighty **Lambdalith**

- The entire application is in **one** Lambda function
- Often times these are “**lift and shift**” Express.js or Flask apps
- Events are **synchronous** via API Gateway or ALB
- **Partial failures** are handled “in the code”



The **Fat Lambda**

The **Fat Lambda**

- Several **related methods** are collocated in a **single** Lambda function

The **Fat Lambda**

- Several **related methods** are collocated in a **single** Lambda function
- Generally used to optimize the speed of **synchronous** operations

The **Fat Lambda**

- Several **related methods** are collocated in a **single** Lambda function
- Generally used to optimize the speed of **synchronous** operations
- **Partial failures** are still handled “in the code”

The **Fat Lambda**

- Several **related methods** are collocated in a **single** Lambda function
- Generally used to optimize the speed of **synchronous** operations
- **Partial failures** are still handled “in the code”
- Under the **right circumstances**, this can be useful

The **Single-Purpose** Function

The **Single-Purpose** Function

- Tightly scoped function that handles a single **discrete piece of business logic**

The **Single-Purpose** Function

- Tightly scoped function that handles a single **discrete piece of business logic**
- Can be invoked **synchronously** or **asynchronously**

The **Single-Purpose** Function

- Tightly scoped function that handles a single **discrete piece of business logic**
- Can be invoked **synchronously** or **asynchronously**
- **Failures** are generally “total failures” and are **passed back** to the invoking service

The **Single-Purpose** Function

- Tightly scoped function that handles a single **discrete piece of business logic**
- Can be invoked **synchronously** or **asynchronously**
- **Failures** are generally “total failures” and are **passed back** to the invoking service
- Can be **reused** as part of other “workflows”, can **scale (or throttle) independently**, and can utilize the **Principle of Least Privilege**

Failure Modes in the Cloud

Failure Modes in the Cloud

WARNING: Firehose of overly-technical content ahead 🧑🚒

Understanding **retries**...

Understanding **retries**...

- Retries are a **vital part** of distributed systems

Understanding **retries**...

- Retries are a **vital part** of distributed systems
- Most cloud services guarantee **"at least once"** delivery

Understanding **retries**...

- Retries are a **vital part** of distributed systems
- Most cloud services guarantee **“at least once”** delivery
- It is possible for the **same event** to be received **more than once**

Understanding **retries**...

- Retries are a **vital part** of distributed systems
- Most cloud services guarantee **“at least once”** delivery
- It is possible for the **same event** to be received **more than once**
- Retried operations should be **idempotent**

Idempotent means that...

Idempotent means that...

An operation can be **repeated multiple times** and always provide the **same result**, with **no side effects** to other objects in the system. ~ *Computer Hope*

Idempotent means that...

An operation can be **repeated multiple times** and always provide the **same result**, with **no side effects** to other objects in the system. ~ *Computer Hope*

Idempotent operations:

Idempotent means that...

An operation can be **repeated multiple times** and always provide the **same result**, with **no side effects** to other objects in the system. ~ *Computer Hope*

Idempotent operations:

- Update a database record

Idempotent means that...

An operation can be **repeated multiple times** and always provide the **same result**, with **no side effects** to other objects in the system. ~ *Computer Hope*

Idempotent operations:

- Update a database record
- Authenticate a user

Idempotent means that...

An operation can be **repeated multiple times** and always provide the **same result**, with **no side effects** to other objects in the system. ~ *Computer Hope*

Idempotent operations:

- Update a database record
- Authenticate a user
- Check if a record exists and create if not

Idempotent means that...

An operation can be **repeated multiple times** and always provide the **same result**, with **no side effects** to other objects in the system. ~ *Computer Hope*

Idempotent operations:

- Update a database record
- Authenticate a user
- Check if a record exists and create if not

There are lots of **strategies** to ensure **idempotency!**

Dead Letter Queues (DLQs)

Dead Letter Queues (DLQs)

- Capture messages/events that **fail to process** or are **skipped**

Dead Letter Queues (DLQs)

- Capture messages/events that **fail to process** or are **skipped**
- Allows for **alarming, inspection,** and potential **replay**

Dead Letter Queues (DLQs)

- Capture messages/events that **fail to process** or are **skipped**
- Allows for **alarming, inspection**, and potential **replay**
- Can be added to **SQS** queues, **SNS** subscriptions, **Lambda** functions

Lambda Invocation Types

Lambda Invocation Types

- **Synchronous** – request/response model

Lambda Invocation Types

- **Synchronous** – request/response model
- **Asynchronous** – set it and forget it

Lambda Invocation Types

- **Synchronous** – request/response model
- **Asynchronous** – set it and forget it
- **Stream-based** – push

Lambda Invocation Types

- **Synchronous** – request/response model
- **Asynchronous** – set it and forget it
- **Stream-based** – push
- **Poller-based** – pull

Synchronous Lambda Retry Behavior

Synchronous Lambda Retry Behavior

- Functions are invoked directly using **request/response** method

Synchronous Lambda Retry Behavior

- Functions are invoked directly using **request/response** method
- Failures are returned to the **invoker**

Synchronous Lambda Retry Behavior

- Functions are invoked directly using **request/response** method
- Failures are returned to the **invoker**
- **Retries** are delegated to the invoking application

Synchronous Lambda Retry Behavior

- Functions are invoked directly using **request/response** method
- Failures are returned to the **invoker**
- **Retries** are delegated to the invoking application
- Some AWS services **automatically retry** (e.g. Alexa & Cognito)

Synchronous Lambda Retry Behavior

- Functions are invoked directly using **request/response** method
- Failures are returned to the **invoker**
- **Retries** are delegated to the invoking application
- Some AWS services **automatically retry** (e.g. Alexa & Cognito)
- Other services **do not retry** (e.g. API Gateway, ALB, Step Functions)

Synchronous Lambda Retry Behavior

- Functions are invoked directly using **request/response** method
- Failures are returned to the **invoker**
- **Retries** are delegated to the invoking application
- Some AWS services **automatically retry** (e.g. Alexa & Cognito)
- Other services **do not retry** (e.g. API Gateway, ALB, Step Functions)
- API Gateway and ALB can return errors to the **client** for retry

Asynchronous Lambda Retry Behavior

Asynchronous Lambda Retry Behavior

- The **Lambda Service** accepts requests and adds them to a **queue**

Asynchronous Lambda Retry Behavior

- The **Lambda Service** accepts requests and adds them to a **queue**
- The invoker receives a 202 status code and **disconnects**

Asynchronous Lambda Retry Behavior

- The **Lambda Service** accepts requests and adds them to a **queue**
- The invoker receives a 202 status code and **disconnects**
- The Lambda Service will attempt to reprocess failed events up to **2 times**, configured using the **MaximumRetryAttempts** setting

Asynchronous Lambda Retry Behavior

- The **Lambda Service** accepts requests and adds them to a **queue**
- The invoker receives a 202 status code and **disconnects**
- The Lambda Service will attempt to reprocess failed events up to **2 times**, configured using the **MaximumRetryAttempts** setting
- If the Lambda function is throttled, the event will be retried for up to **6 hours**, configured using **MaximumEventAgeInSeconds**

Asynchronous Lambda Retry Behavior

- The **Lambda Service** accepts requests and adds them to a **queue**
- The invoker receives a 202 status code and **disconnects**
- The Lambda Service will attempt to reprocess failed events up to **2 times**, configured using the **MaximumRetryAttempts** setting
- If the Lambda function is throttled, the event will be retried for up to **6 hours**, configured using **MaximumEventAgeInSeconds**
- Failed and expired events can be sent to a **Dead Letter Queue** (DLQ) or an **on-failure destination**

Stream-based Lambda Retry Behavior

Stream-based Lambda Retry Behavior

- Records are pushed **synchronously** to **Lambda** from Kinesis or DynamoDB streams in batches (10k and 1k limits per batch)

Stream-based Lambda Retry Behavior

- Records are pushed **synchronously** to **Lambda** from Kinesis or DynamoDB streams in batches (10k and 1k limits per batch)
- **MaximumRetryAttempts**: number of retry attempts for batches before they can be skipped (up to 10,000)

Stream-based Lambda Retry Behavior

- Records are pushed **synchronously** to **Lambda** from Kinesis or DynamoDB streams in batches (10k and 1k limits per batch)
- **MaximumRetryAttempts:** number of retry attempts for batches before they can be skipped (up to 10,000)
- **MaximumRecordAgeInSeconds:** store records up to **7 days**

Stream-based Lambda Retry Behavior

- Records are pushed **synchronously** to **Lambda** from Kinesis or DynamoDB streams in batches (10k and 1k limits per batch)
- **MaximumRetryAttempts:** number of retry attempts for batches before they can be skipped (up to 10,000)
- **MaximumRecordAgeInSeconds:** store records up to **7 days**
- **BisectBatchOnFunctionError:** recursively split failed batches (poison pill)

Stream-based Lambda Retry Behavior

- Records are pushed **synchronously** to **Lambda** from Kinesis or DynamoDB streams in batches (10k and 1k limits per batch)
- **MaximumRetryAttempts**: number of retry attempts for batches before they can be skipped (up to 10,000)
- **MaximumRecordAgeInSeconds**: store records up to **7 days**
- **BisectBatchOnFunctionError**: recursively split failed batches (poison pill)
- Skipped records are sent to an **On-failure Destination** (SQS or SNS)

Poller-based Lambda Retry Behavior

Poller-based Lambda Retry Behavior

- The **Lambda Poller** pulls records **synchronously** from SQS in batches (up to 10)

Poller-based Lambda Retry Behavior

- The **Lambda Poller** pulls records **synchronously** from SQS in batches (up to 10)
- Errors fail the **entire batch**

Poller-based Lambda Retry Behavior

- The **Lambda Poller** pulls records **synchronously** from SQS in batches (up to 10)
- Errors fail the **entire batch**
- **MaxReceiveCount**: number of times messages can be returned to the queue before being sent to the DLQ (up to 1,000)

Poller-based Lambda Retry Behavior

- The **Lambda Poller** pulls records **synchronously** from SQS in batches (up to 10)
- Errors fail the **entire batch**
- **MaxReceiveCount**: number of times messages can be returned to the queue before being sent to the DLQ (up to 1,000)
- Polling **frequency** is tied to function **concurrency**

Poller-based Lambda Retry Behavior

- The **Lambda Poller** pulls records **synchronously** from SQS in batches (up to 10)
- Errors fail the **entire batch**
- **MaxReceiveCount**: number of times messages can be returned to the queue before being sent to the DLQ (up to 1,000)
- Polling **frequency** is tied to function **concurrency**
- **Visibility Timeout** should be set to at least **6 times** the timeout configured on your consuming function

Lambda Destinations

Lambda Destinations

- Only for **asynchronous** invocations

Lambda Destinations

- Only for **asynchronous** invocations
- Routing based on **SUCCESS** and/or **FAILURE**

Lambda Destinations

- Only for **asynchronous** invocations
- Routing based on **SUCCESS** and/or **FAILURE**
- **OnFailure** should be favored over a standard **DLQ**

Lambda Destinations

- Only for **asynchronous** invocations
- Routing based on **SUCCESS** and/or **FAILURE**
- **OnFailure** should be favored over a standard **DLQ**
- Destinations can be an **SQS queue**, **SNS topic**, **Lambda function**, or **EventBridge** event bus

Lambda Destinations (continued)

Lambda Destinations (continued)

Destination-specific JSON format

Lambda Destinations (continued)

Destination-specific JSON format

- **SQS/SNS:** JSON object is passed as the *Message*

Lambda Destinations (continued)

Destination-specific JSON format

- **SQS/SNS:** JSON object is passed as the *Message*
- **Lambda:** JSON is passed as the payload to the function

Lambda Destinations (continued)

Destination-specific JSON format

- **SQS/SNS:** JSON object is passed as the *Message*
- **Lambda:** JSON is passed as the payload to the function
- **EventBridge:** JSON is passed as the *Detail* in the PutEvents call

Lambda Destinations (continued)

Destination-specific JSON format

- **SQS/SNS:** JSON object is passed as the *Message*
- **Lambda:** JSON is passed as the payload to the function
- **EventBridge:** JSON is passed as the *Detail* in the PutEvents call
 - Source is "lambda"
 - Detail Type is "Lambda Function Invocation Result – Success/Failure"
 - Resource fields contain the function and destination ARNs

SQS Redrive Policies

SQS Redrive Policies

- Only supports another **SQS queue** as the DLQ

SQS Redrive Policies

- Only supports another **SQS queue** as the DLQ
- Messages are sent to the DLQ if the **Maximum Receives** value is exceeded

SNS Redrive Policies

SNS Redrive Policies

- Dead Letter Queues are attached to **Subscriptions**, not **Topics**

SNS Redrive Policies

- Dead Letter Queues are attached to **Subscriptions**, not **Topics**
- Only supports **SQS queues** as the DLQ

SNS Redrive Policies

- Dead Letter Queues are attached to **Subscriptions**, not **Topics**
- Only supports **SQS queues** as the DLQ
- Client-side errors (e.g. Lambda doesn't exist) **do no retry**

SNS Redrive Policies

- Dead Letter Queues are attached to **Subscriptions**, not **Topics**
- Only supports **SQS queues** as the DLQ
- Client-side errors (e.g. Lambda doesn't exist) **do no retry**
- Messages to **SQS** or **Lambda** are retried **100,015** times over **23** days

SNS Redrive Policies

- Dead Letter Queues are attached to **Subscriptions**, not **Topics**
- Only supports **SQS queues** as the DLQ
- Client-side errors (e.g. Lambda doesn't exist) **do no retry**
- Messages to **SQS** or **Lambda** are retried **100,015** times over **23** days
- Messages to **SMTP**, **SMS**, and **Mobile** retry **50** times over **6** hours

SNS Redrive Policies

- Dead Letter Queues are attached to **Subscriptions**, not **Topics**
- Only supports **SQS queues** as the DLQ
- Client-side errors (e.g. Lambda doesn't exist) **do no retry**
- Messages to **SQS** or **Lambda** are retried **100,015** times over **23** days
- Messages to **SMTP**, **SMS**, and **Mobile** retry **50** times over **6** hours
- **HTTP endpoints** support customer-defined retry policies (number of retries, delays, and backoff strategy)

EventBridge Retry Behavior

EventBridge Retry Behavior

- Will attempt to deliver events for up to **24 hours** with backoff

EventBridge Retry Behavior

- Will attempt to deliver events for up to **24 hours** with backoff
- **Failed events** are lost (this is very unlikely)

EventBridge Retry Behavior

- Will attempt to deliver events for up to **24 hours** with backoff
- **Failed events** are lost (this is very unlikely)
- Once events are accepted by the target service, **failure modes** of those services are used

EventBridge Retry Behavior

- Will attempt to deliver events for up to **24 hours** with backoff
- **Failed events** are lost (this is very unlikely)
- Once events are accepted by the target service, **failure modes** of those services are used
- Lambda functions are invoked **asynchronously**

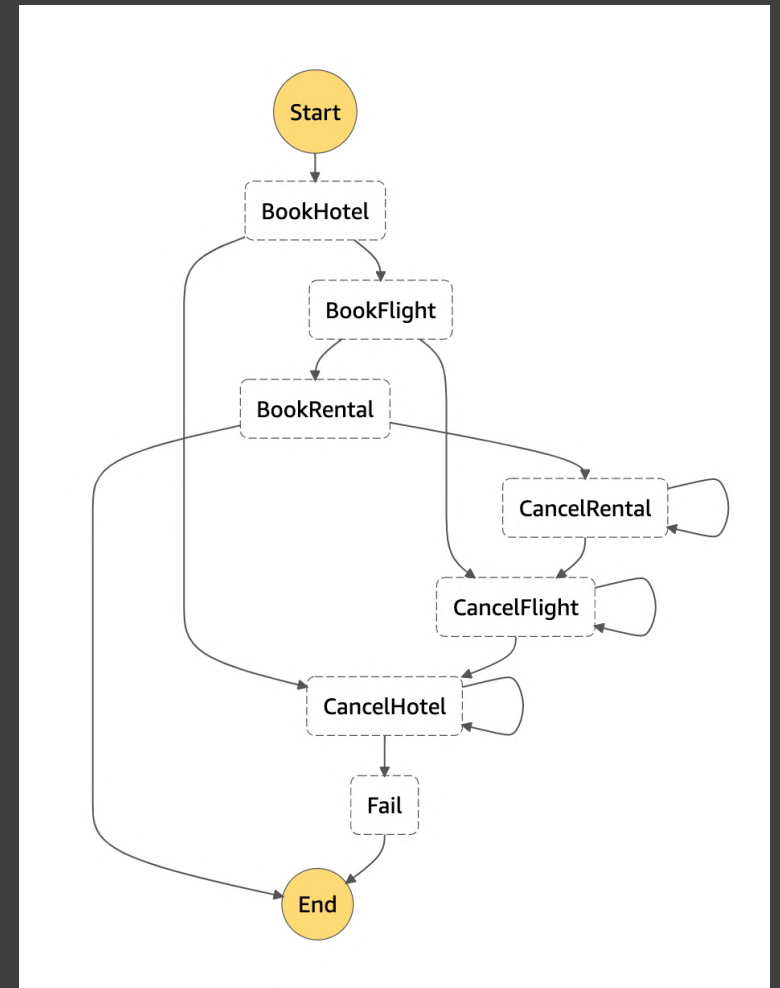
Step Functions

Step Functions

- **State Machines:** Orchestration workflows

Step Functions

- **State Machines:** Orchestration workflows

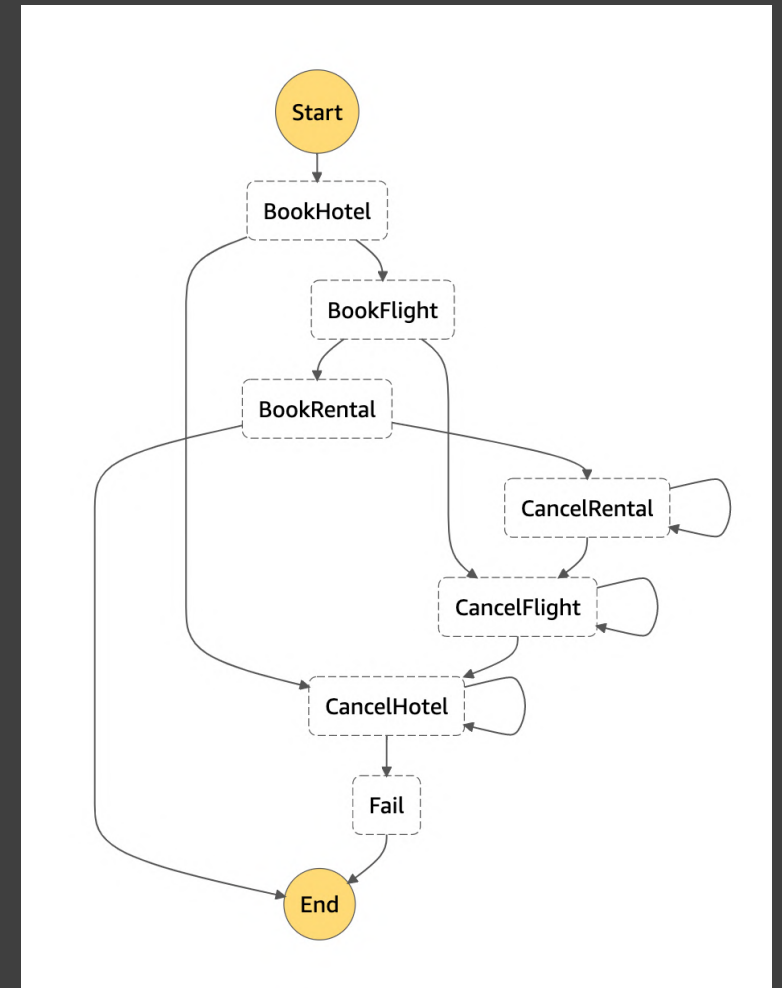


Complex Error Handling Pattern

Credit: Yan Cui

Step Functions

- **State Machines:** Orchestration workflows
- Lambdas are invoked **synchronously**

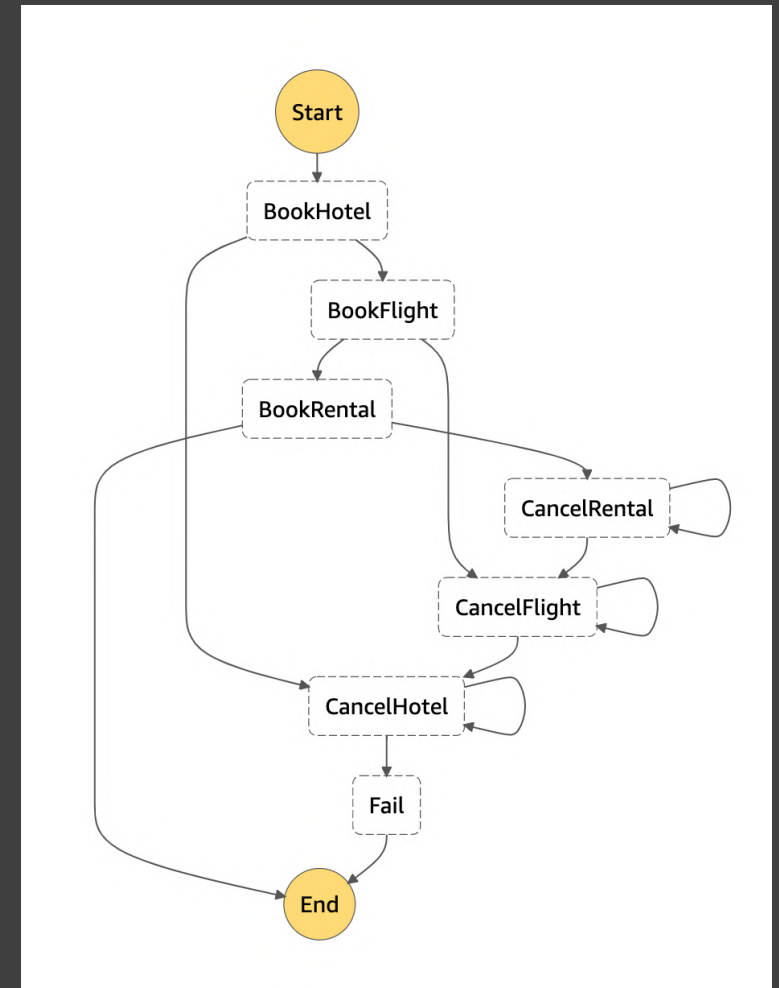


Complex Error Handling Pattern

Credit: Yan Cui

Step Functions

- **State Machines:** Orchestration workflows
- Lambdas are invoked **synchronously**
- **Retriers** and **Catchers** allow for complex error handling patterns

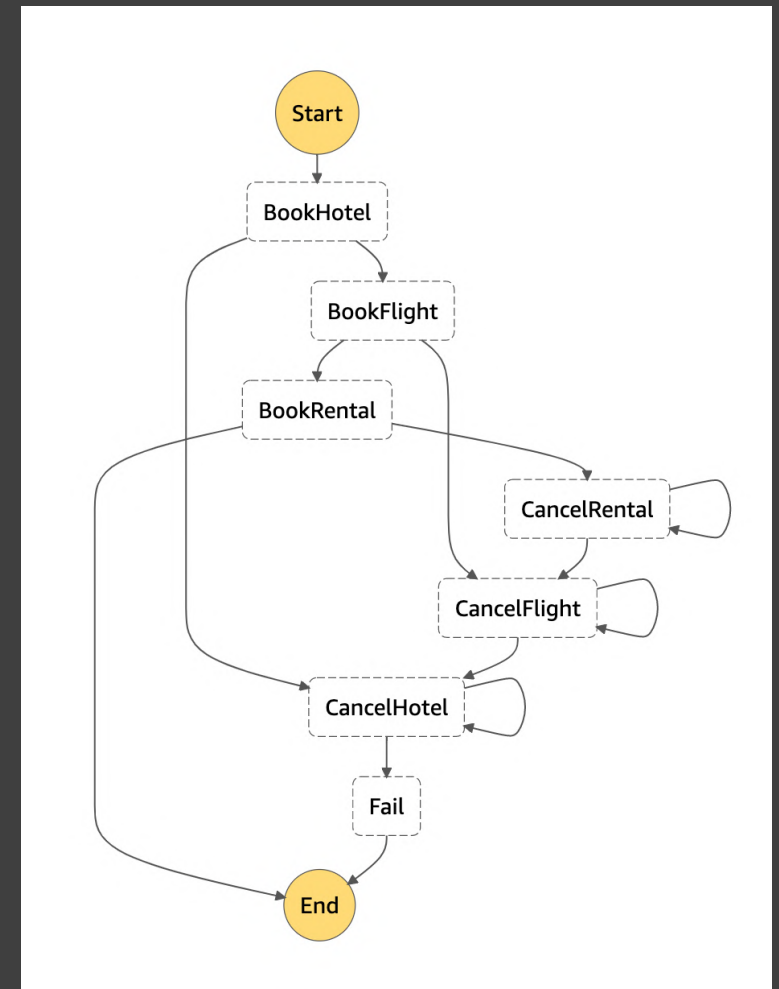


Complex Error Handling Pattern

Credit: Yan Cui

Step Functions

- **State Machines:** Orchestration workflows
- Lambdas are invoked **synchronously**
- **Retriers** and **Catchers** allow for complex error handling patterns
- Use “error names” with *ErrorEquals* for **condition error handling** (States.*)

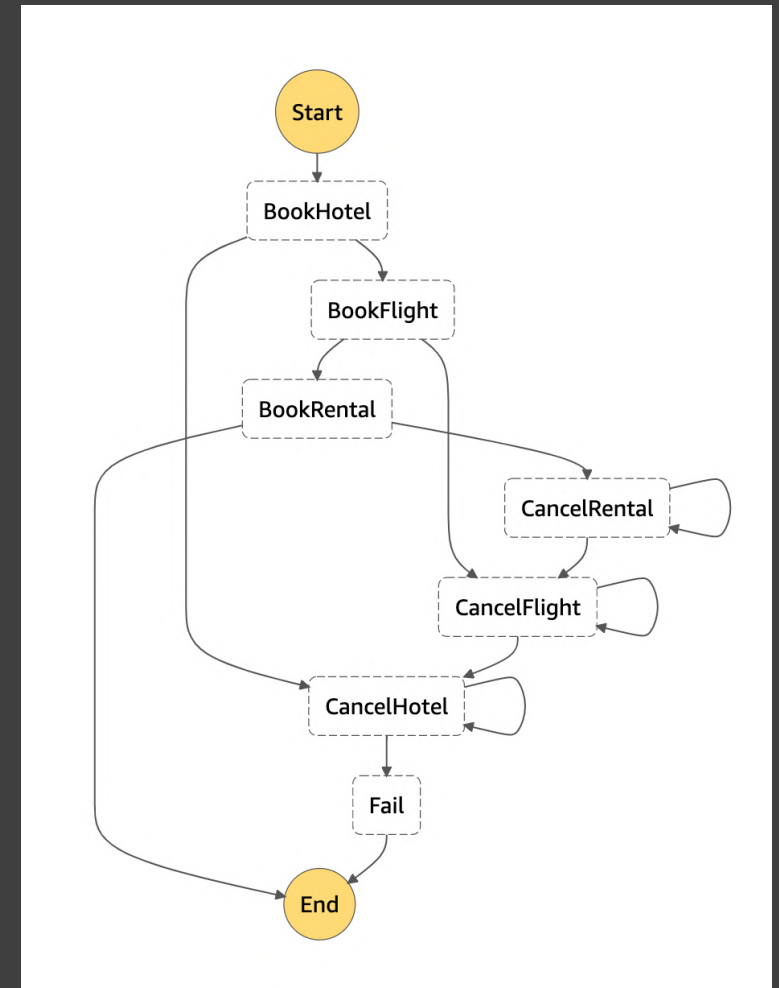


Complex Error Handling Pattern

Credit: Yan Cui

Step Functions

- **State Machines:** Orchestration workflows
- Lambdas are invoked **synchronously**
- **Retriers** and **Catchers** allow for complex error handling patterns
- Use “error names” with *ErrorEquals* for **condition error handling** (*States.**)
- Control **retry policies** with *IntervalSeconds*, *MaxAttempts*, *BackoffRate*



Complex Error Handling Pattern

Credit: Yan Cui

AWS SDK Retries

AWS SDK Retries

- Automatic retries and **exponential backoff**

AWS SDK Retries

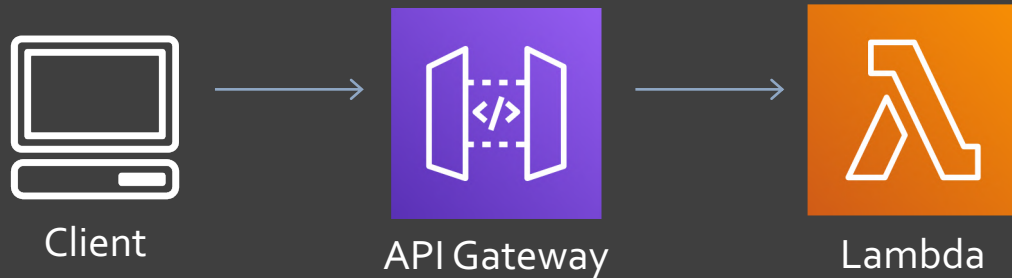
- Automatic retries and **exponential backoff**

AWS SDK	Maximum retry count	Connection timeout	Socket timeout
Python (Boto 3)	depends on service	60 seconds	60 seconds
Node.js	depends on service	N/A	120 seconds
Java	3	10 seconds	50 seconds
.NET	4	100 seconds	300 seconds
Go	3	N/A	N/A

Error Handling Patterns

Buffer events for **throttling** and **durability**

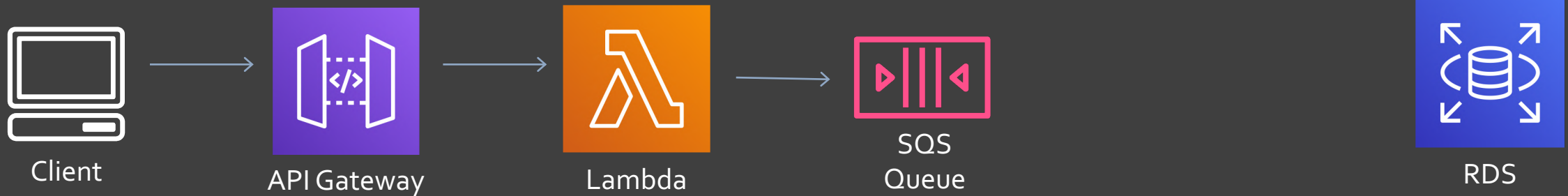
Buffer events for **throttling** and **durability**



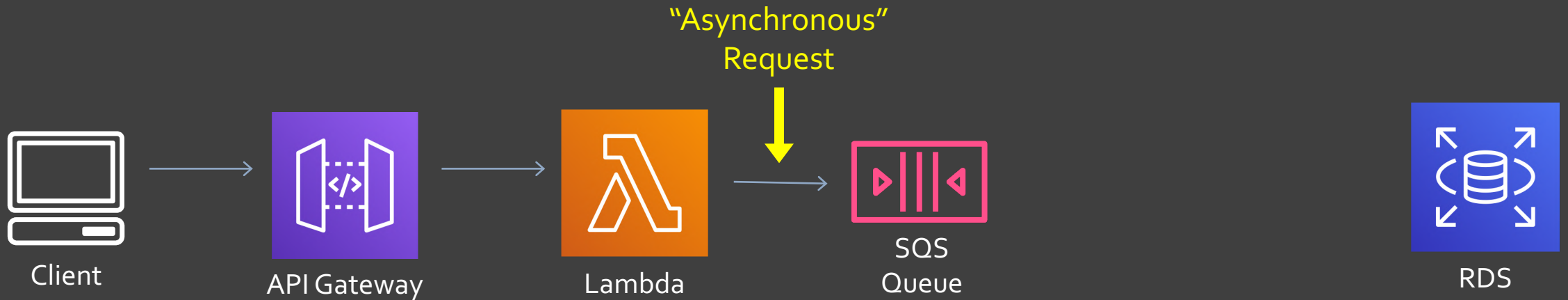
Buffer events for **throttling** and **durability**



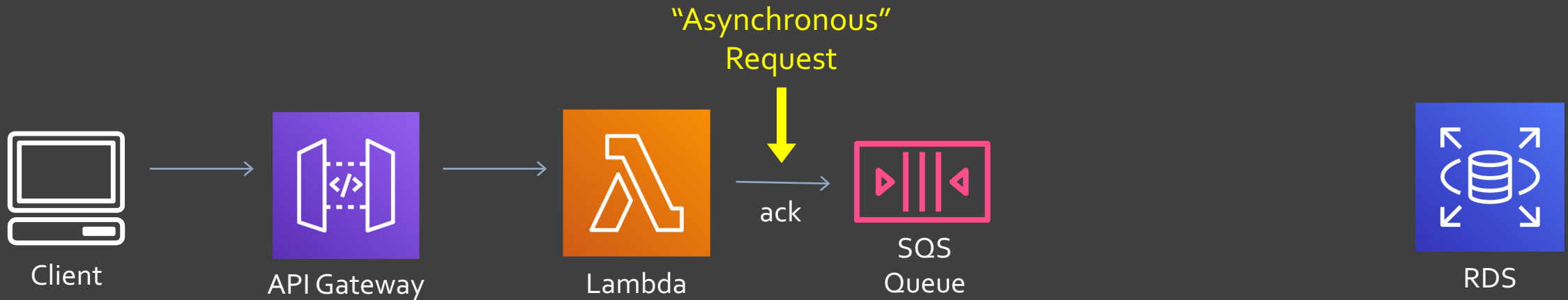
Buffer events for **throttling** and **durability**



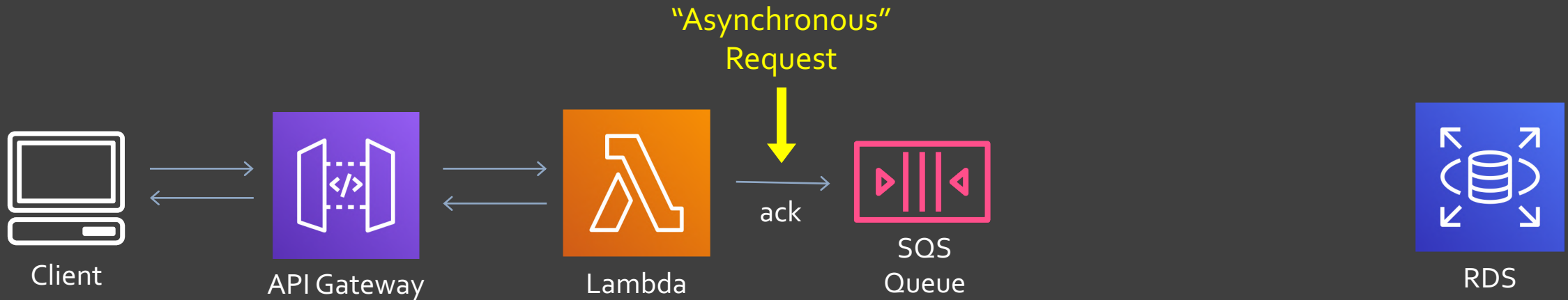
Buffer events for **throttling** and **durability**



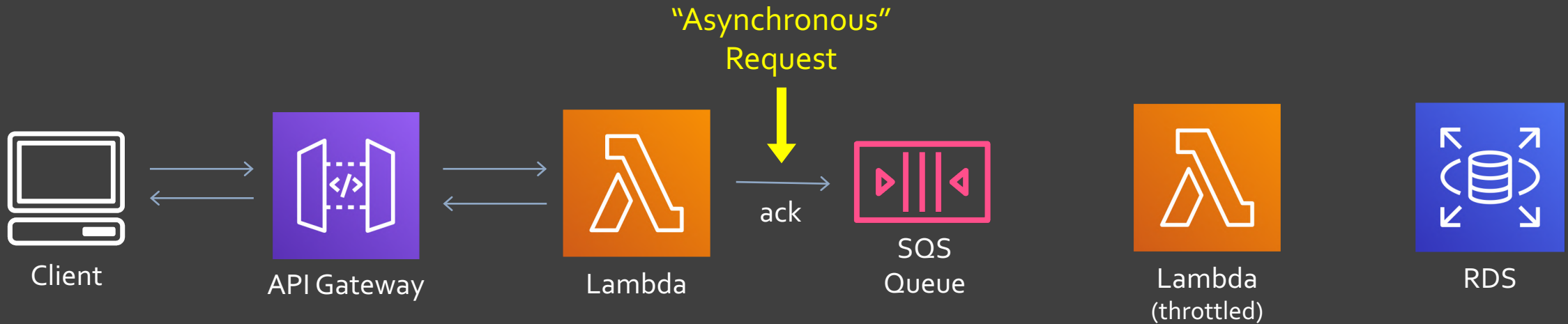
Buffer events for **throttling** and **durability**



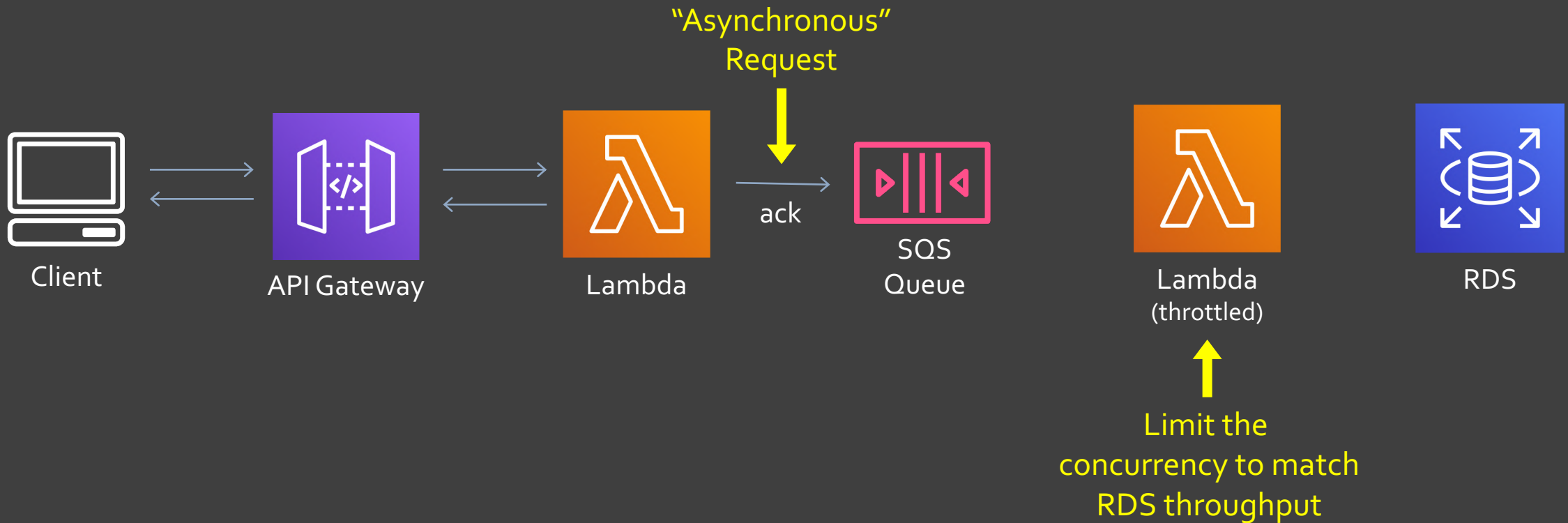
Buffer events for **throttling** and **durability**



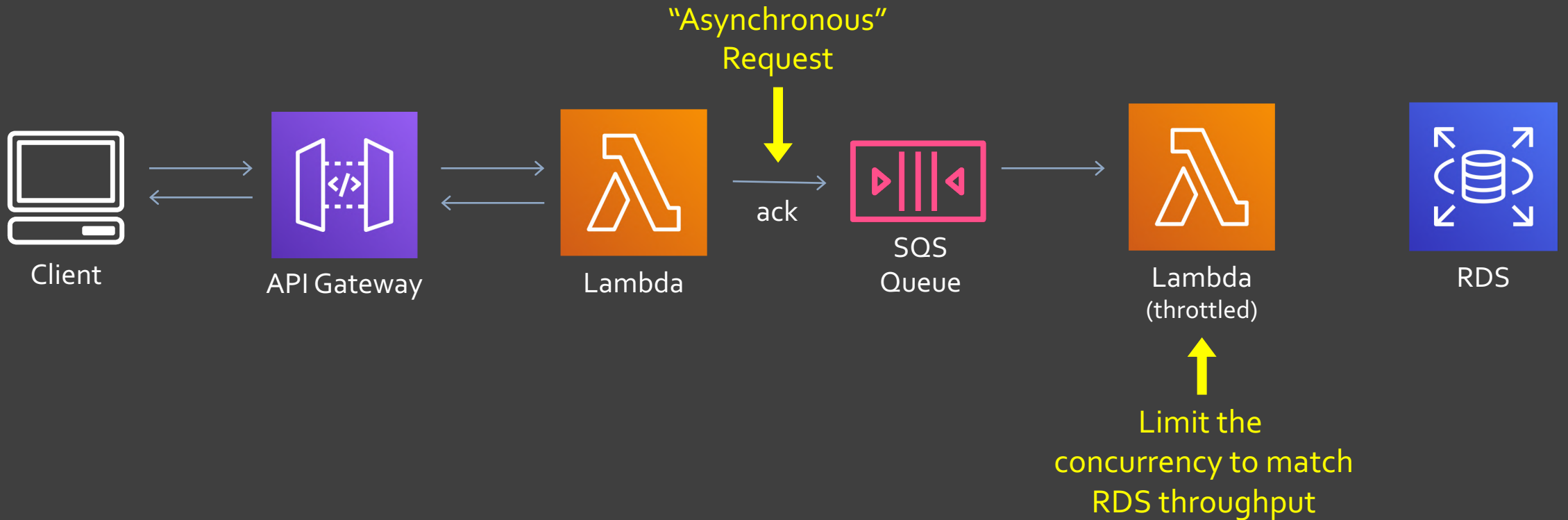
Buffer events for **throttling** and **durability**



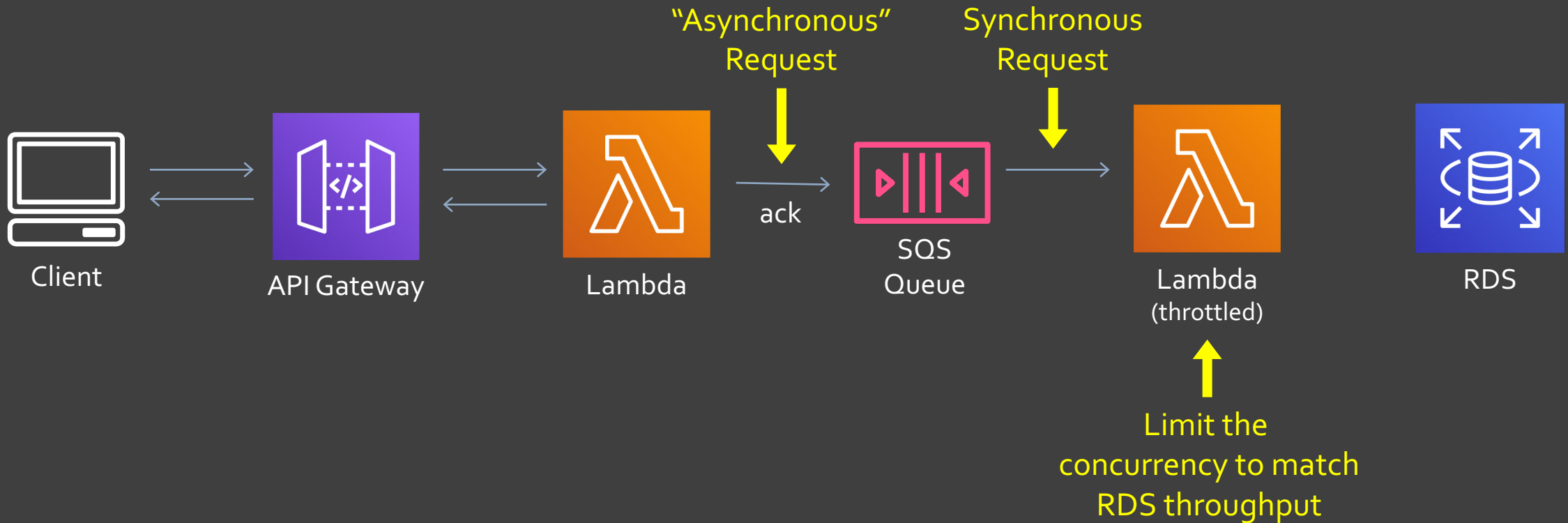
Buffer events for **throttling** and **durability**



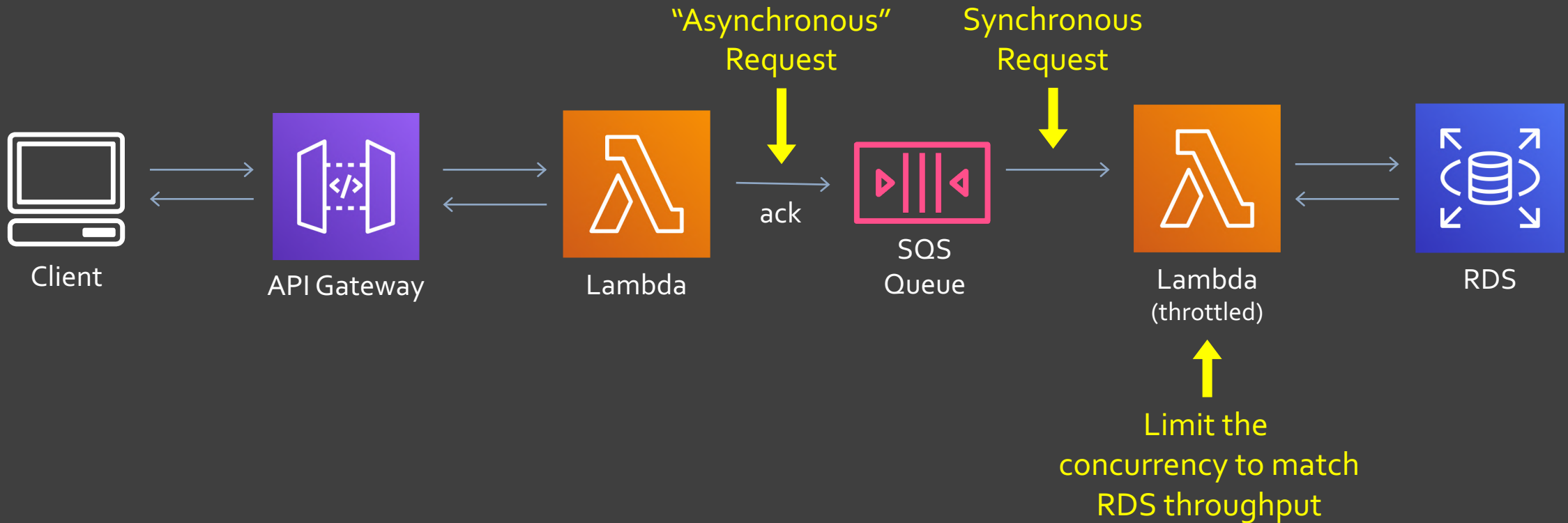
Buffer events for **throttling** and **durability**



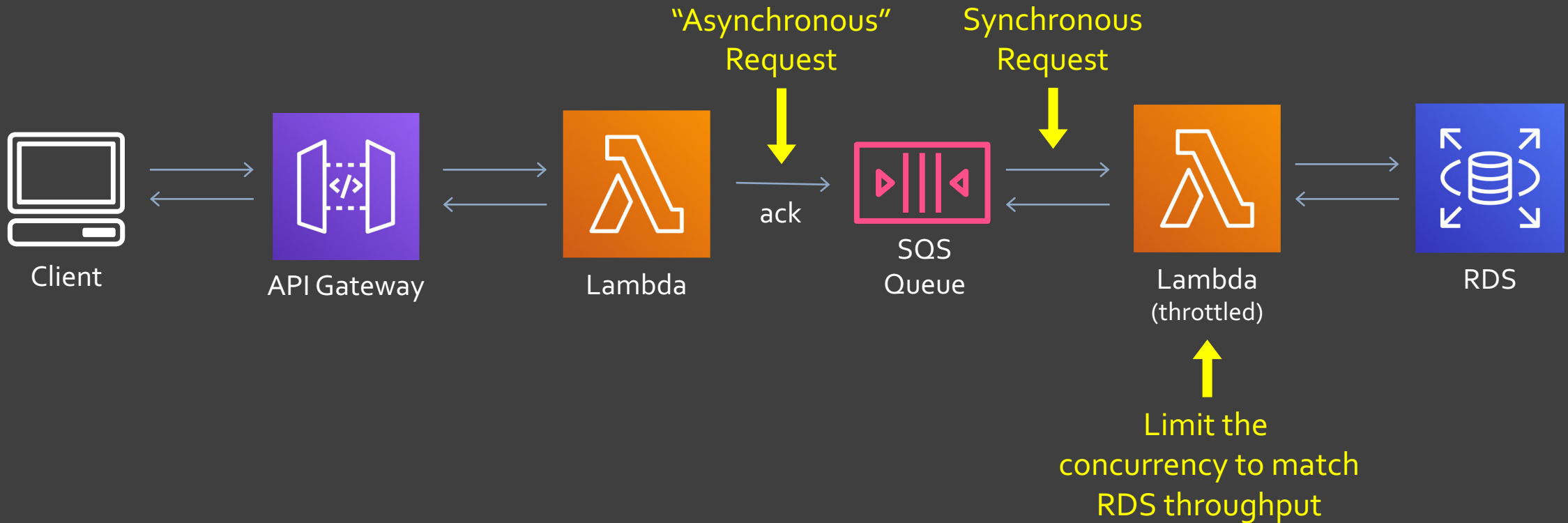
Buffer events for **throttling** and **durability**



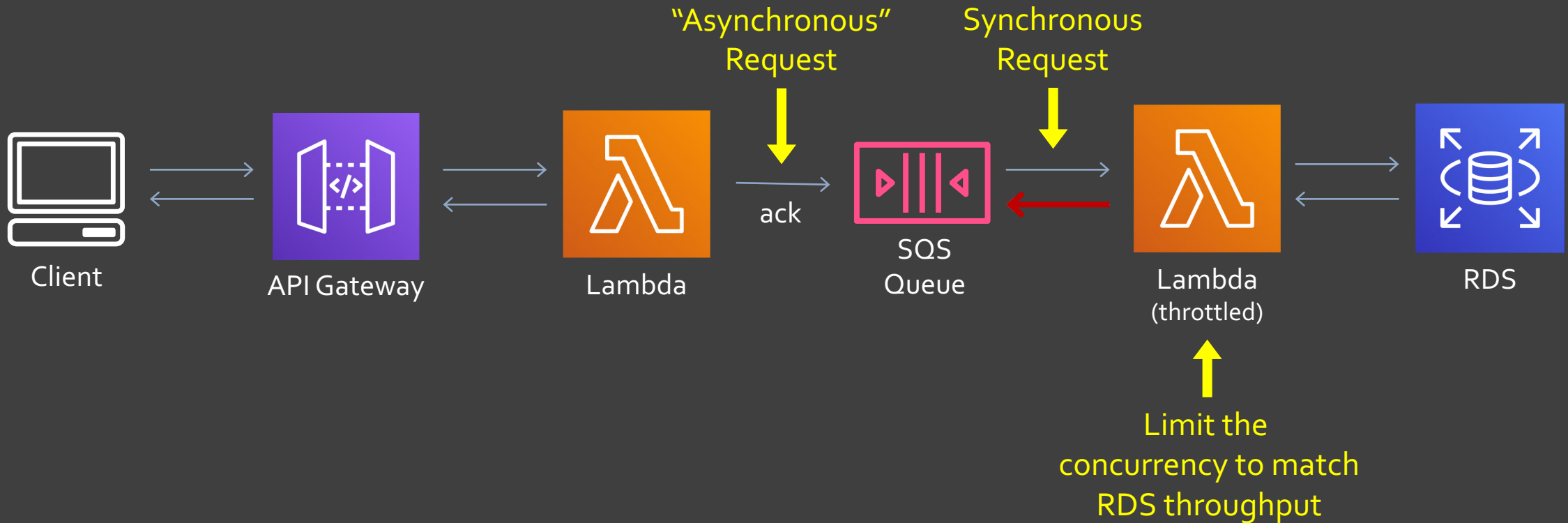
Buffer events for **throttling** and **durability**



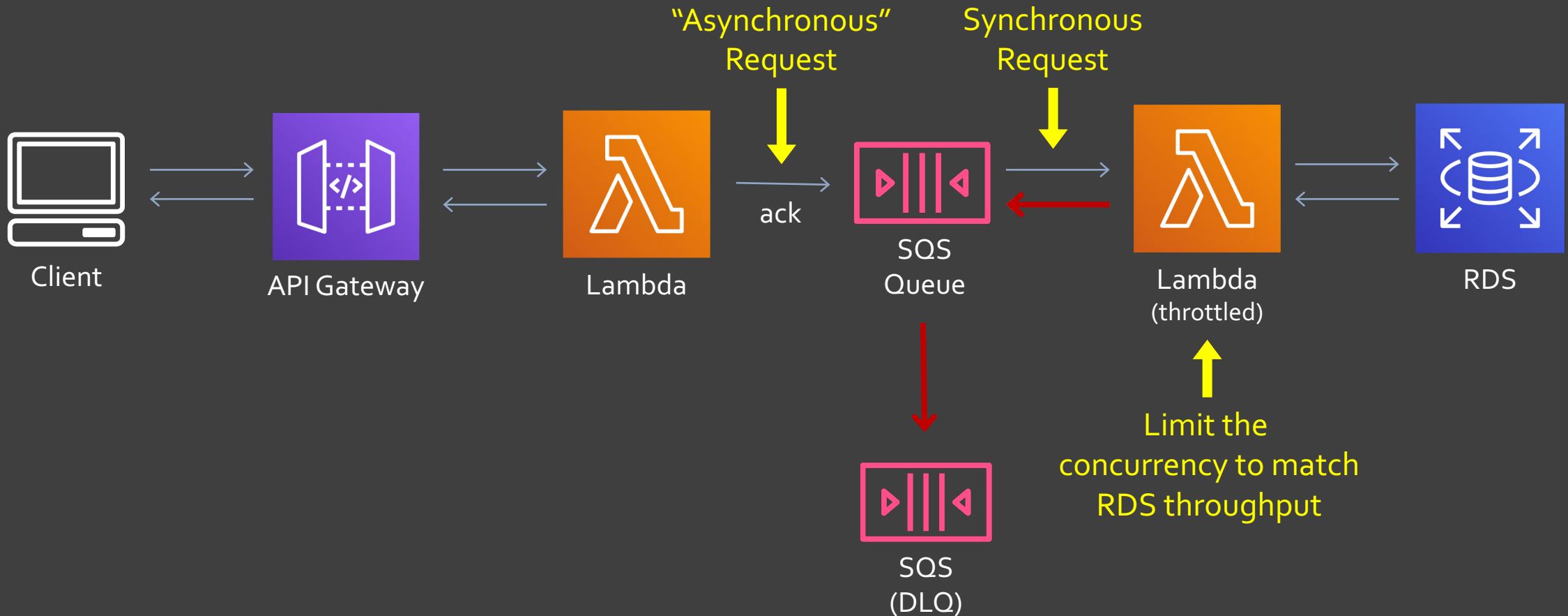
Buffer events for **throttling** and **durability**



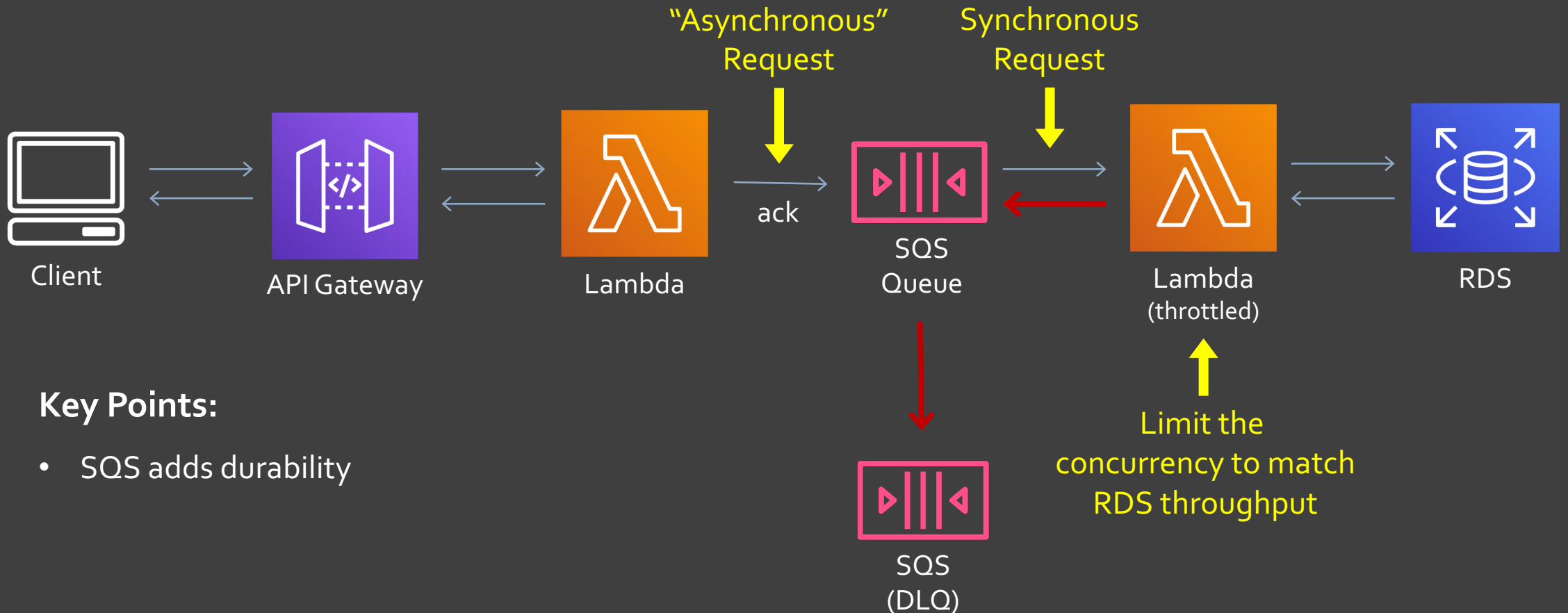
Buffer events for **throttling** and **durability**



Buffer events for **throttling** and **durability**



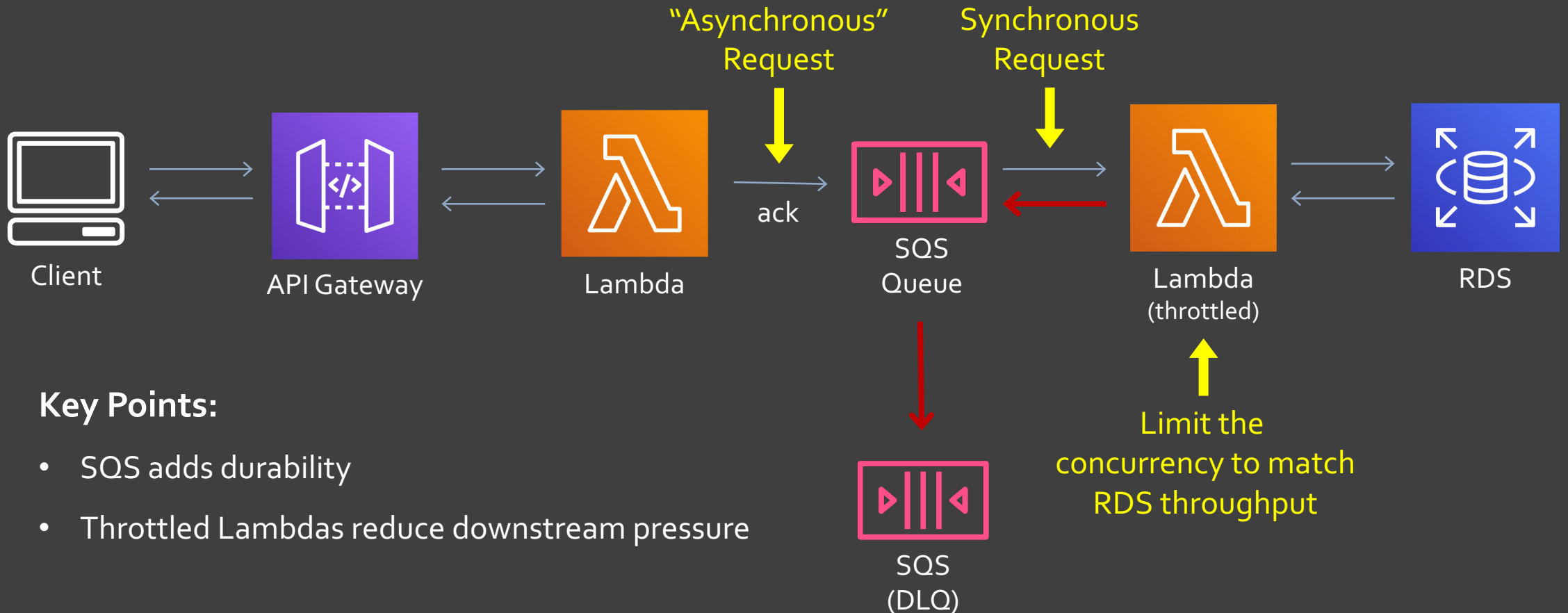
Buffer events for **throttling** and **durability**



Key Points:

- SQS adds durability

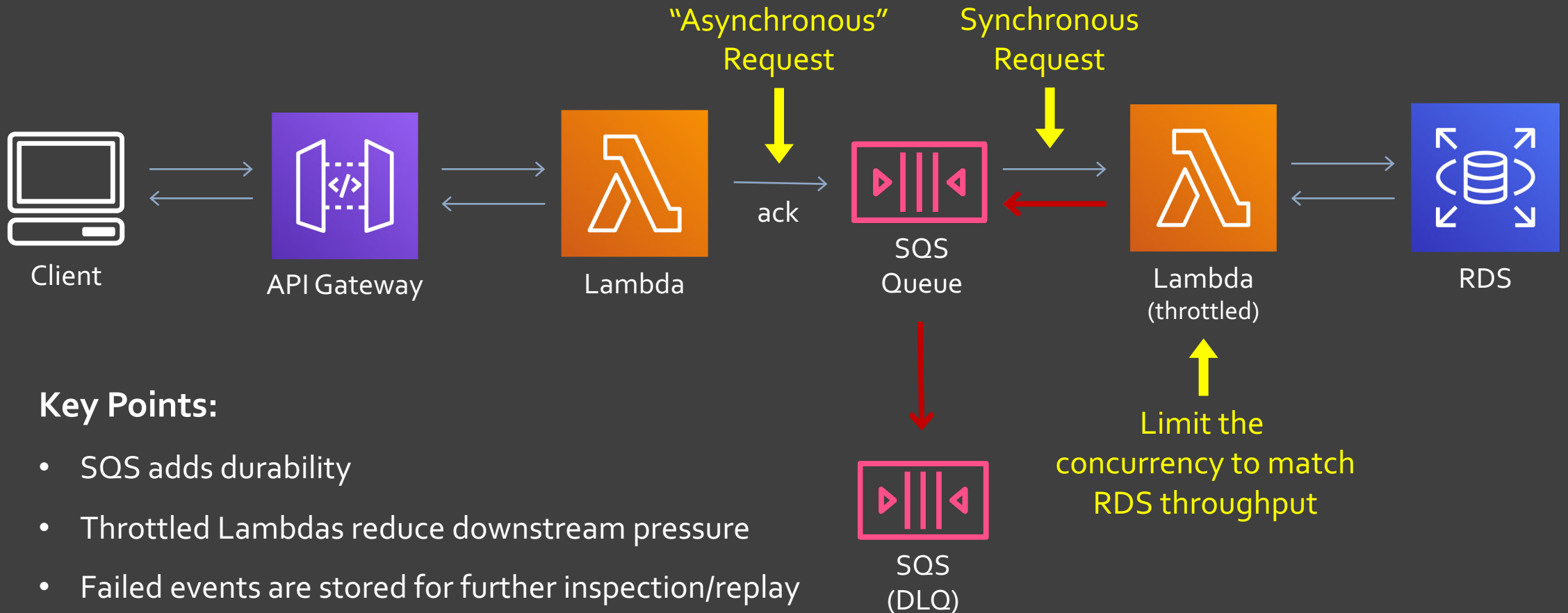
Buffer events for **throttling** and **durability**



Key Points:

- SQS adds durability
- Throttled Lambdas reduce downstream pressure

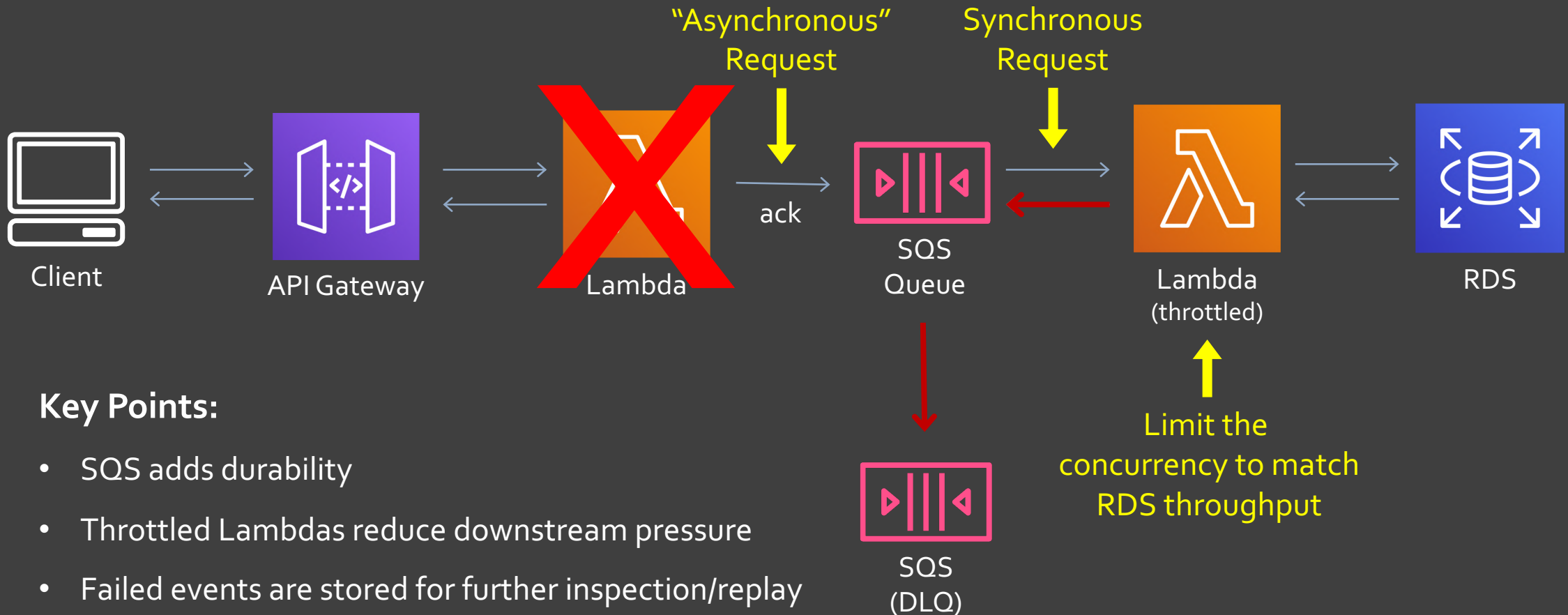
Buffer events for **throttling** and **durability**



Key Points:

- SQS adds durability
- Throttled Lambdas reduce downstream pressure
- Failed events are stored for further inspection/replay

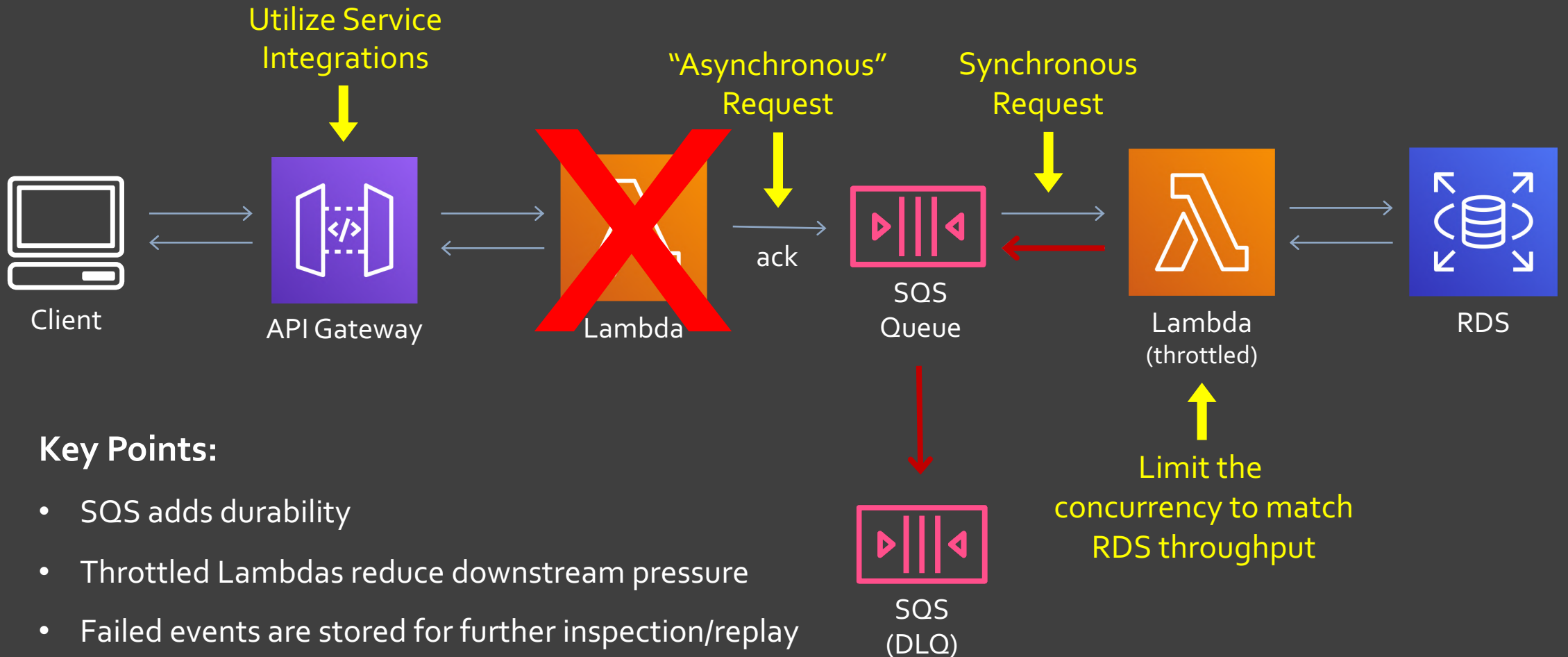
Buffer events for **throttling** and **durability**



Key Points:

- SQS adds durability
- Throttled Lambdas reduce downstream pressure
- Failed events are stored for further inspection/replay

Buffer events for **throttling** and **durability**

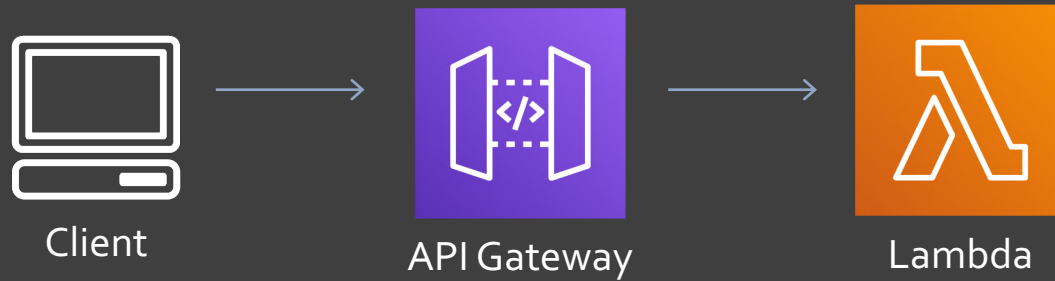


Key Points:

- SQS adds durability
- Throttled Lambdas reduce downstream pressure
- Failed events are stored for further inspection/replay

The Circuit Breaker

The Circuit Breaker



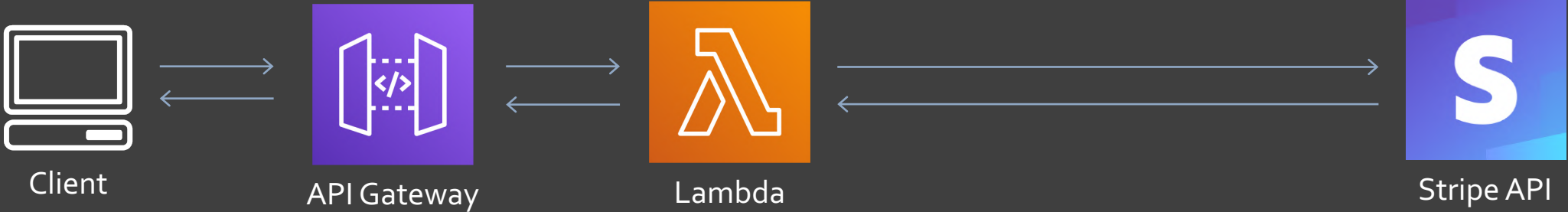
The Circuit Breaker



The Circuit Breaker



The Circuit Breaker



The Circuit Breaker

"Everything fails all the time."
~ Werner Vogels

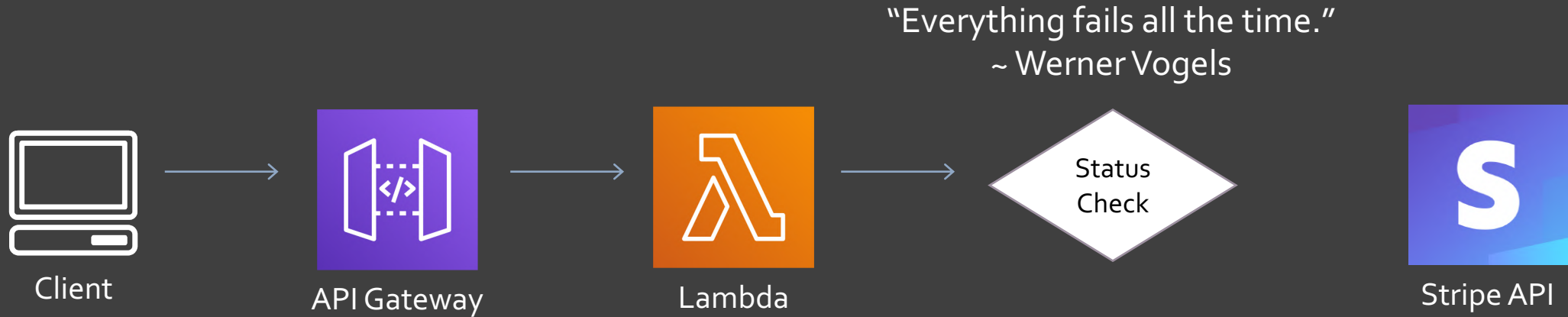


The Circuit Breaker

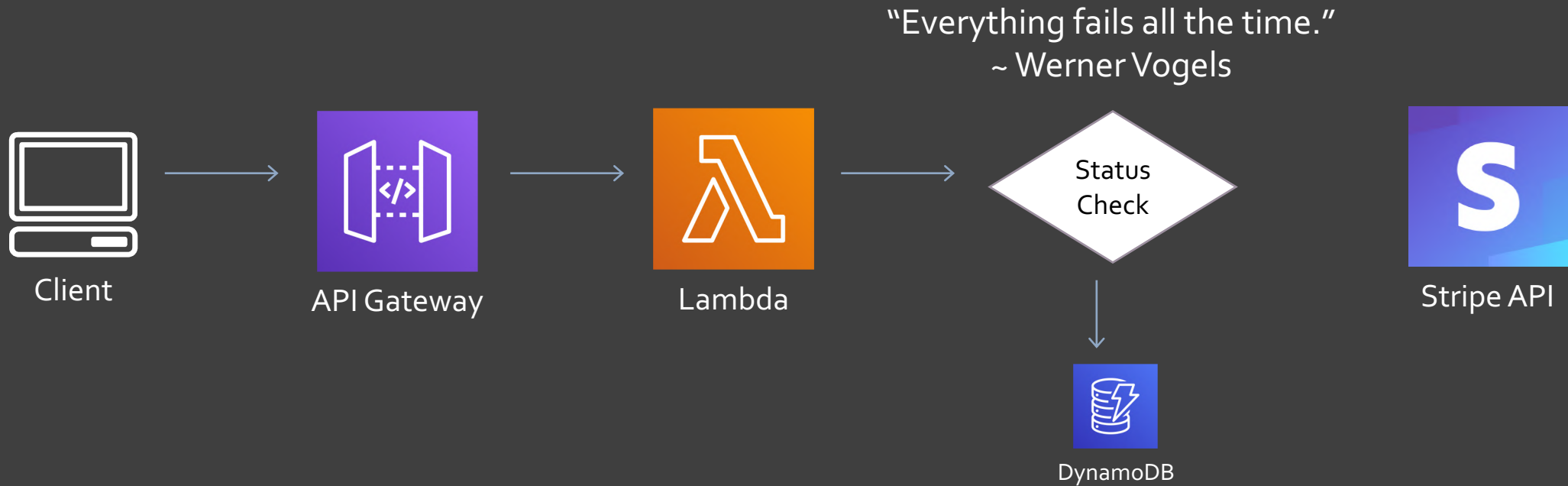
"Everything fails all the time."
~ Werner Vogels



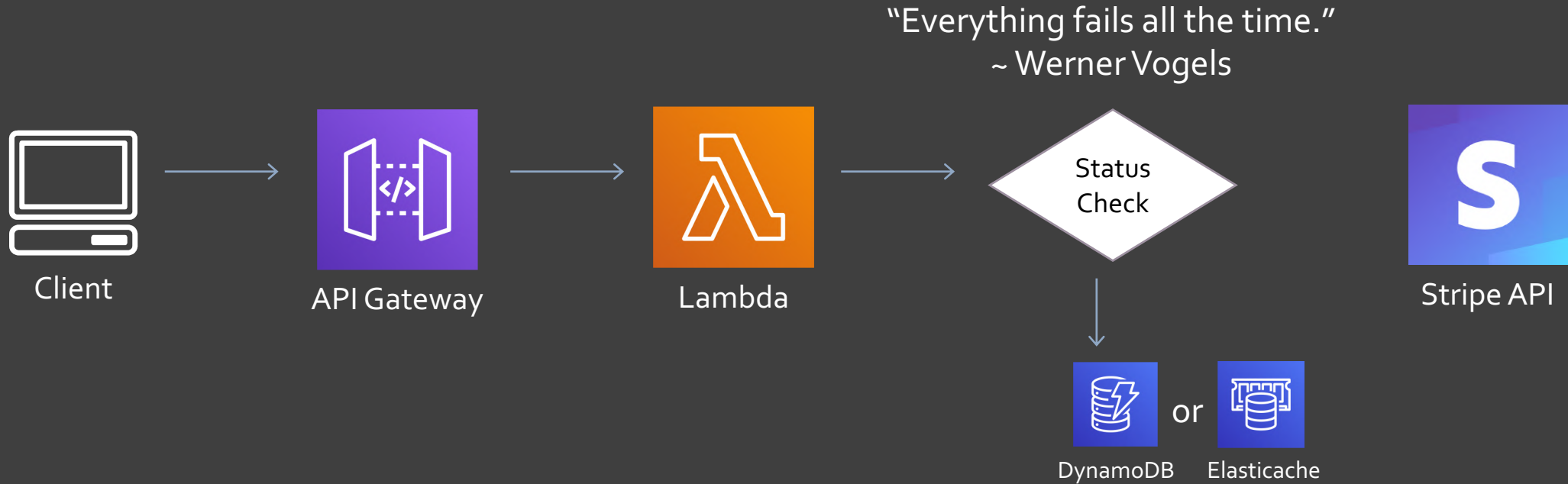
The Circuit Breaker



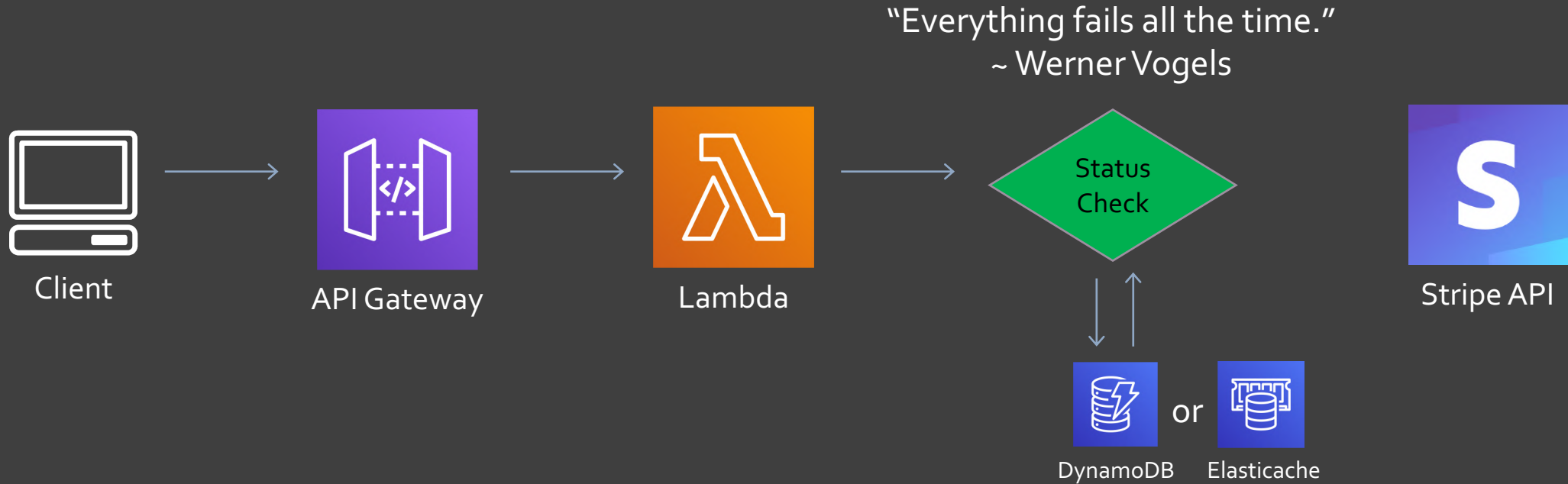
The Circuit Breaker



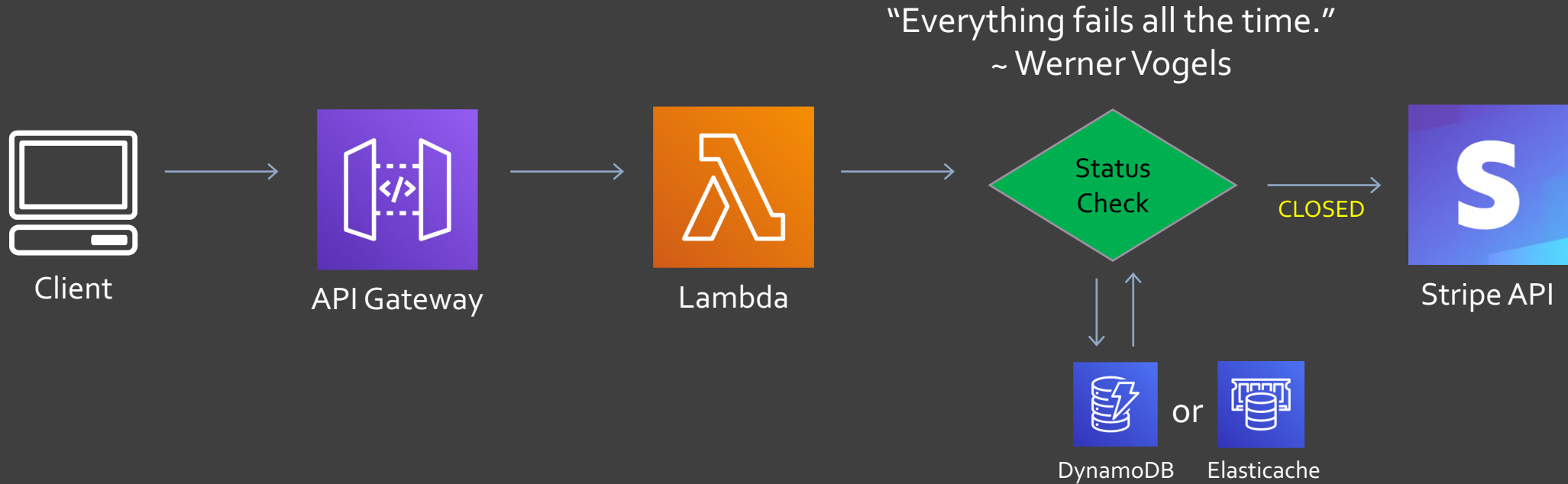
The Circuit Breaker



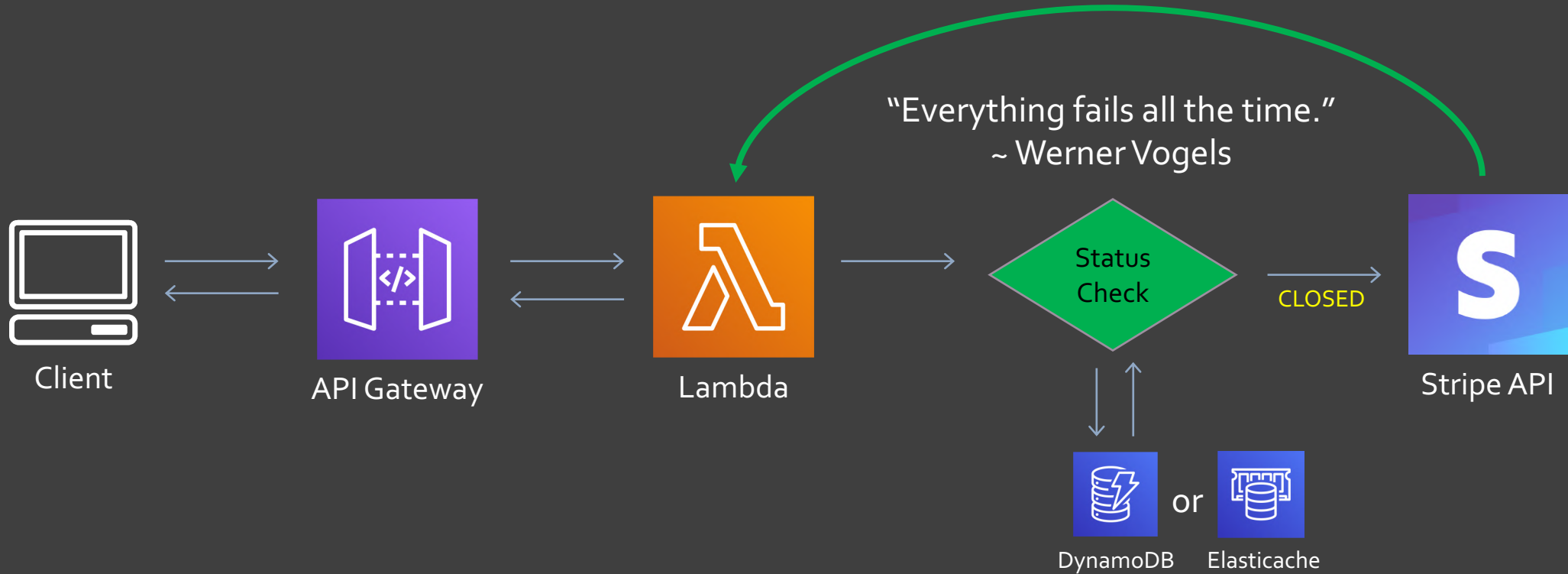
The Circuit Breaker



The Circuit Breaker

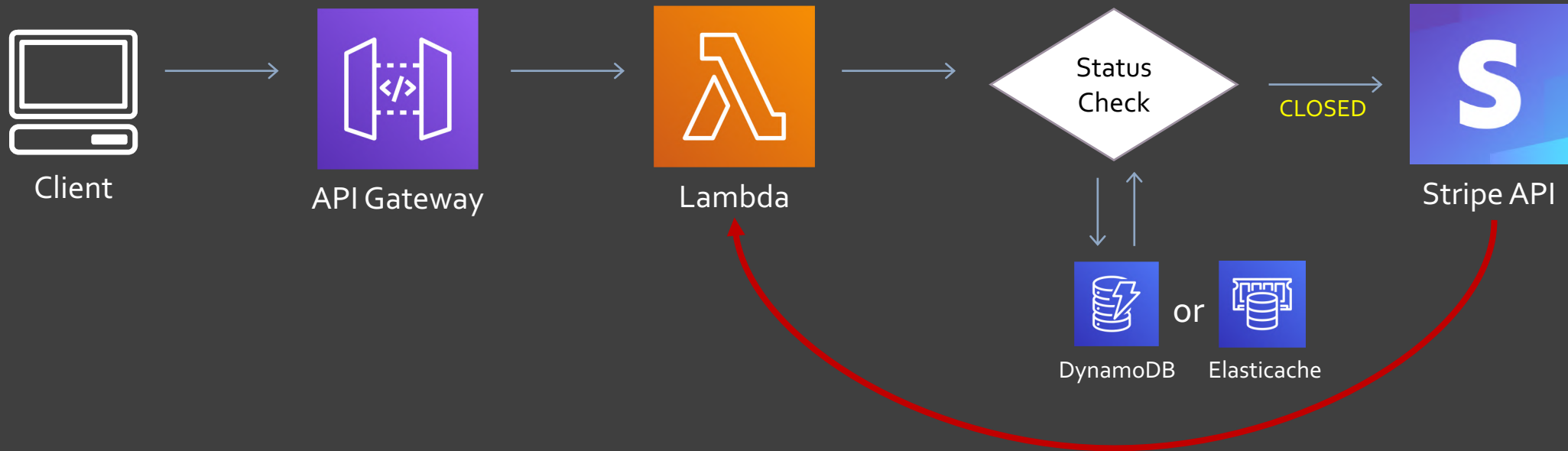


The Circuit Breaker



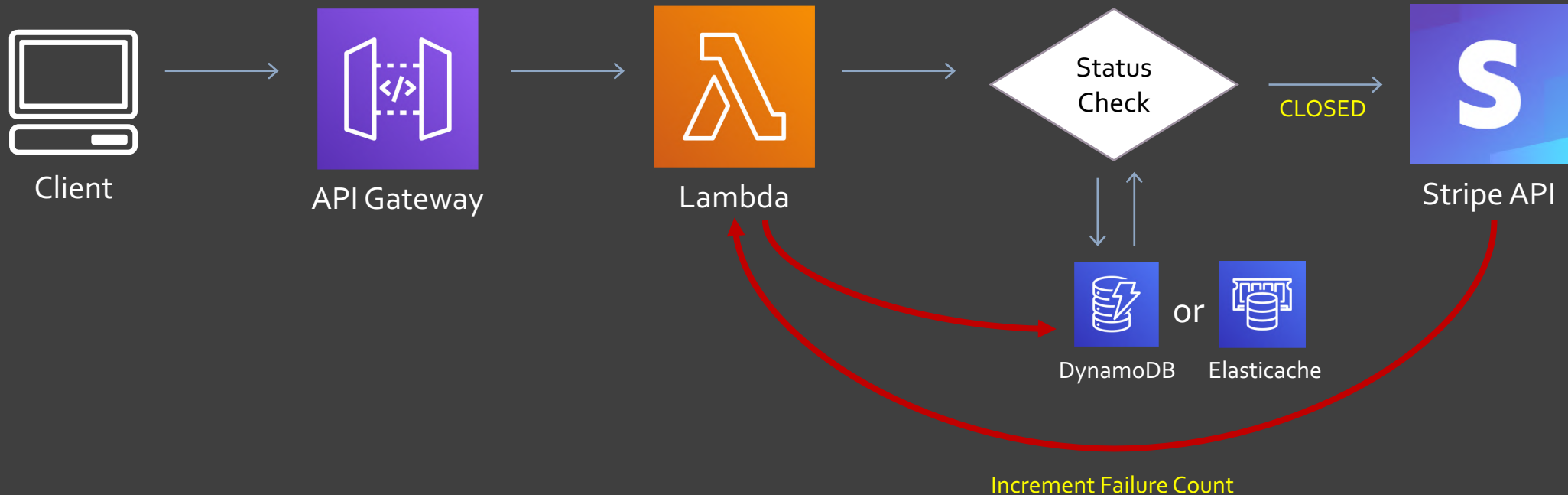
The Circuit Breaker

"Everything fails all the time."
~ Werner Vogels



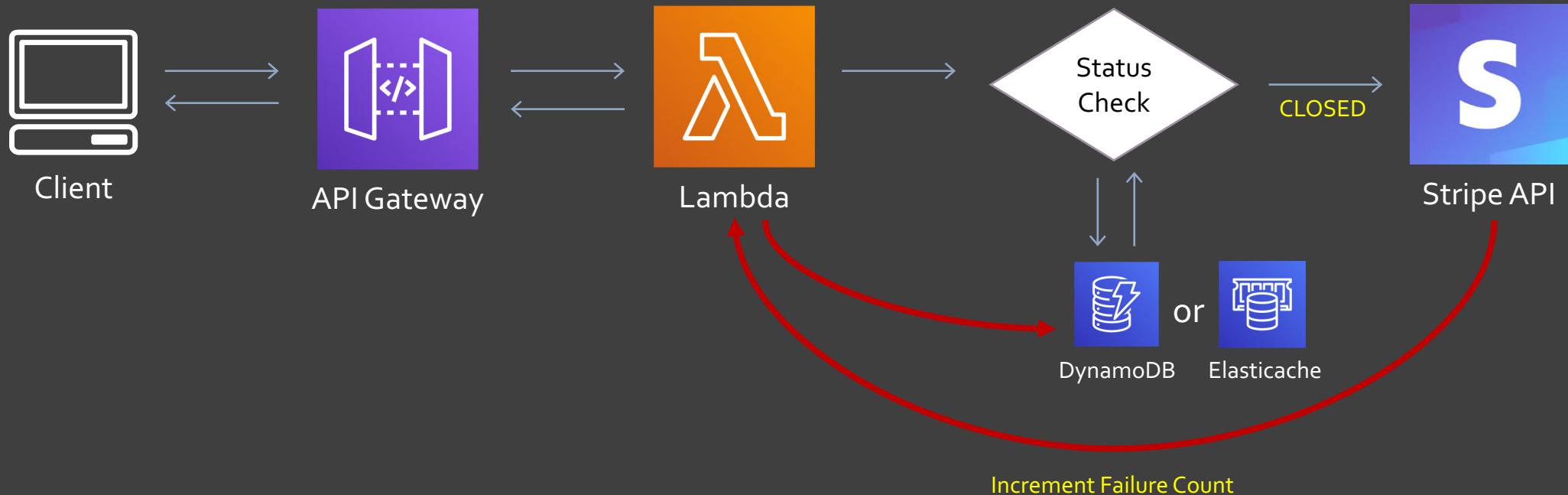
The Circuit Breaker

"Everything fails all the time."
~ Werner Vogels



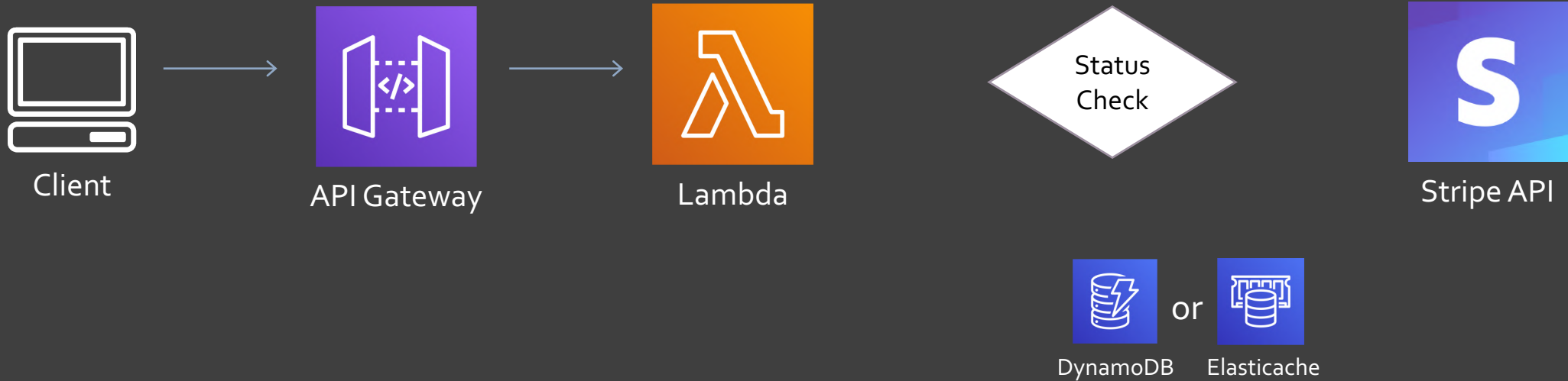
The Circuit Breaker

"Everything fails all the time."
~ Werner Vogels

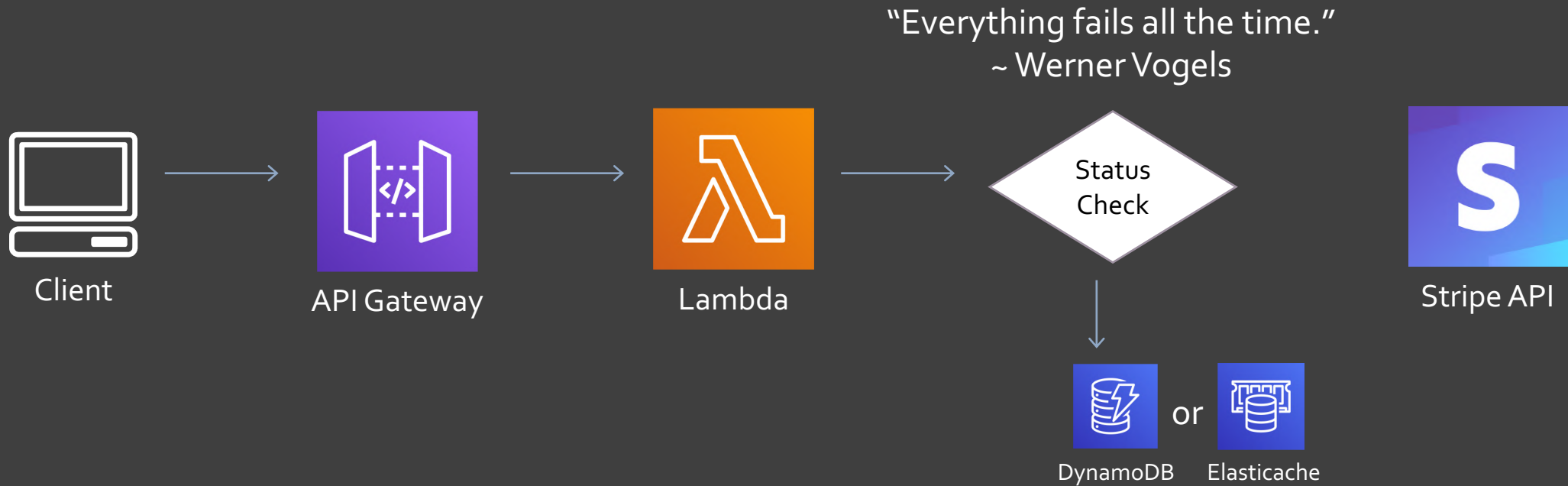


The Circuit Breaker

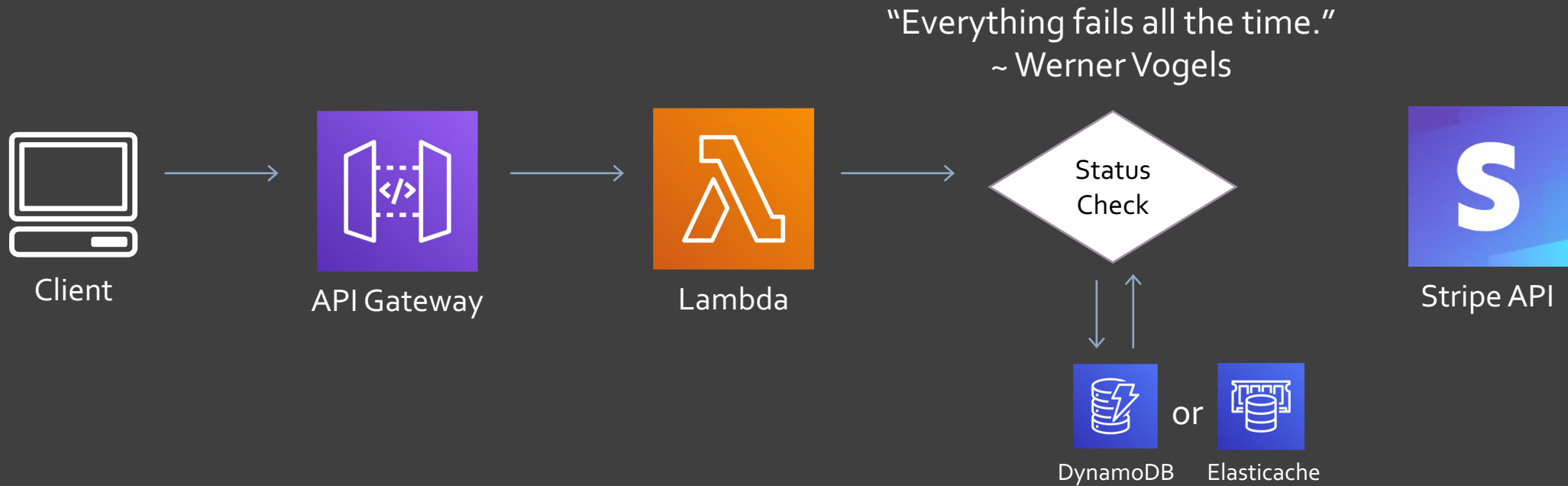
"Everything fails all the time."
~ Werner Vogels



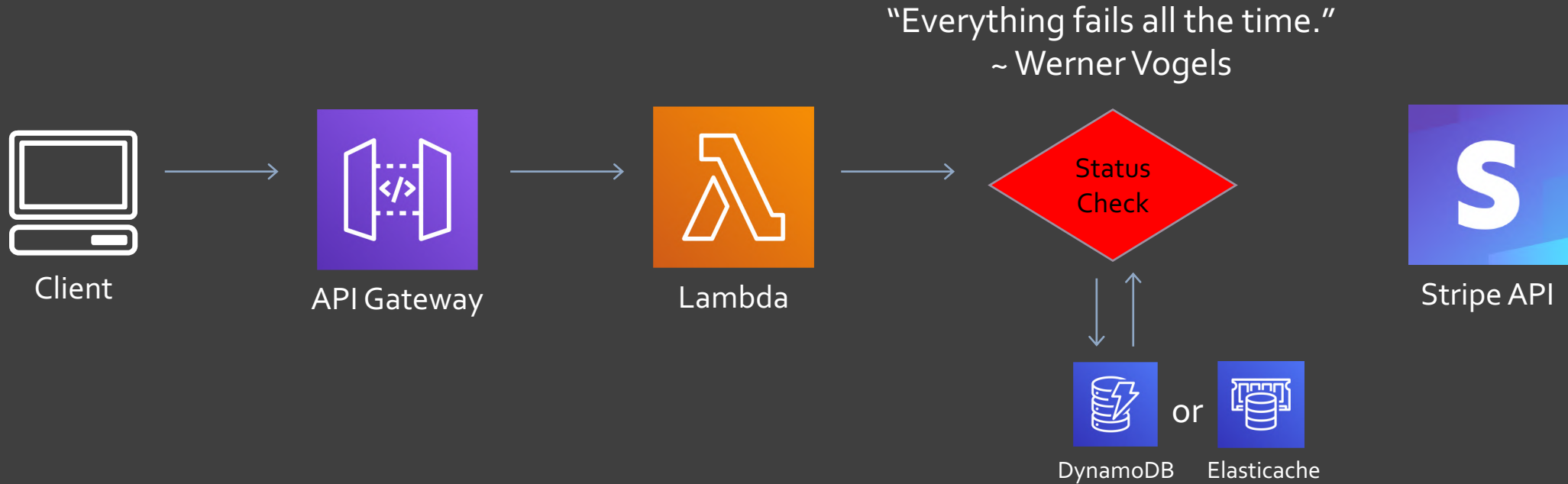
The Circuit Breaker



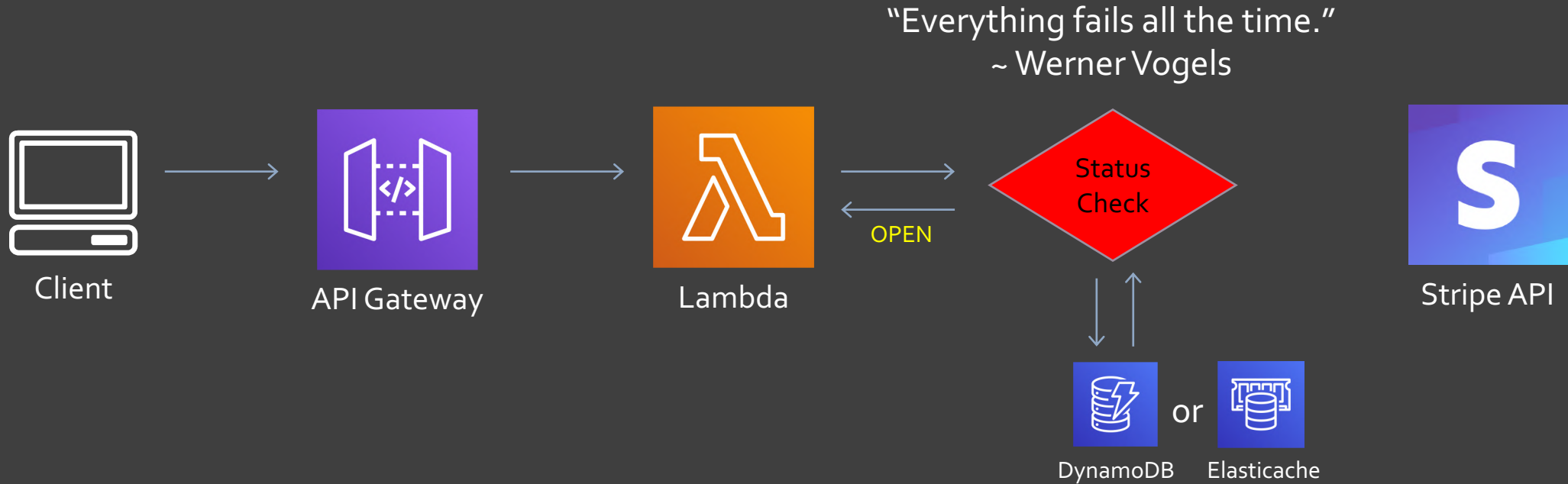
The Circuit Breaker



The Circuit Breaker

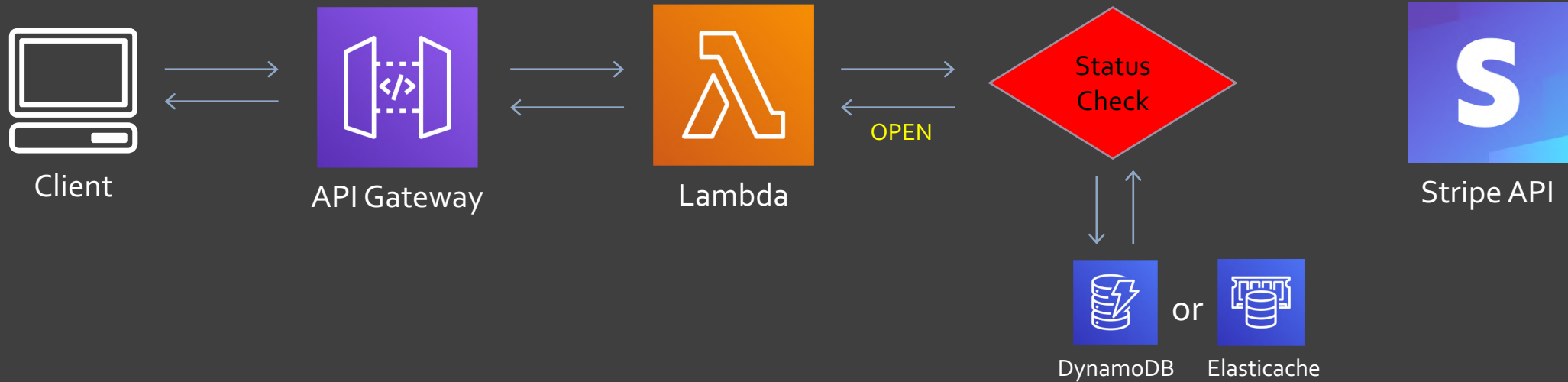


The Circuit Breaker

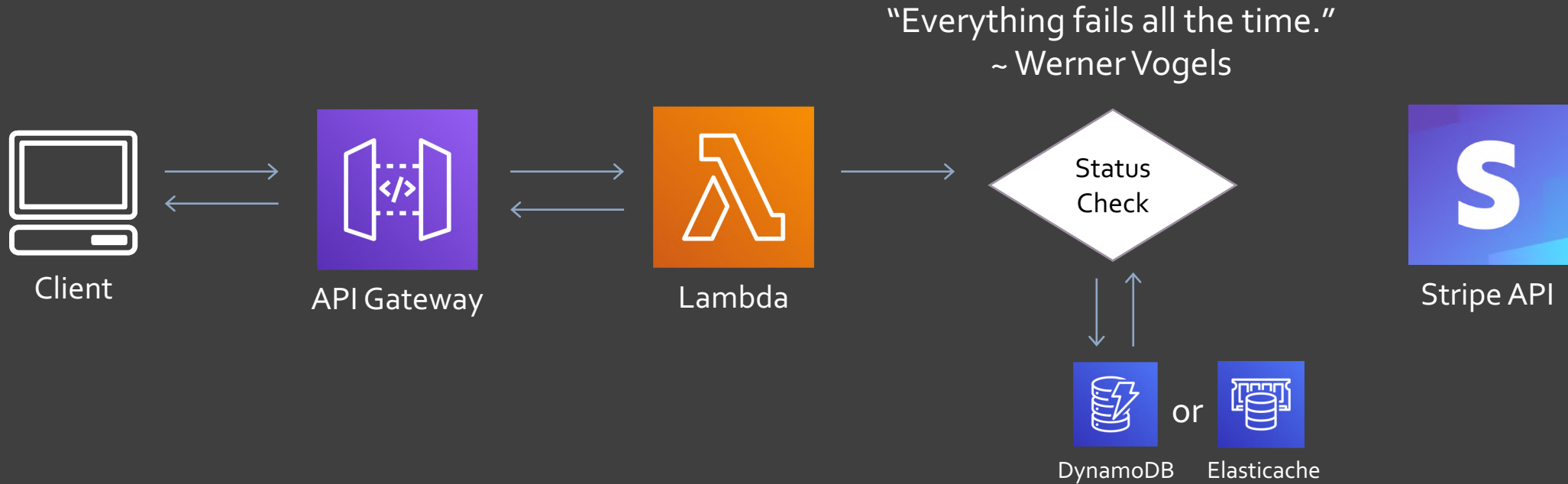


The Circuit Breaker

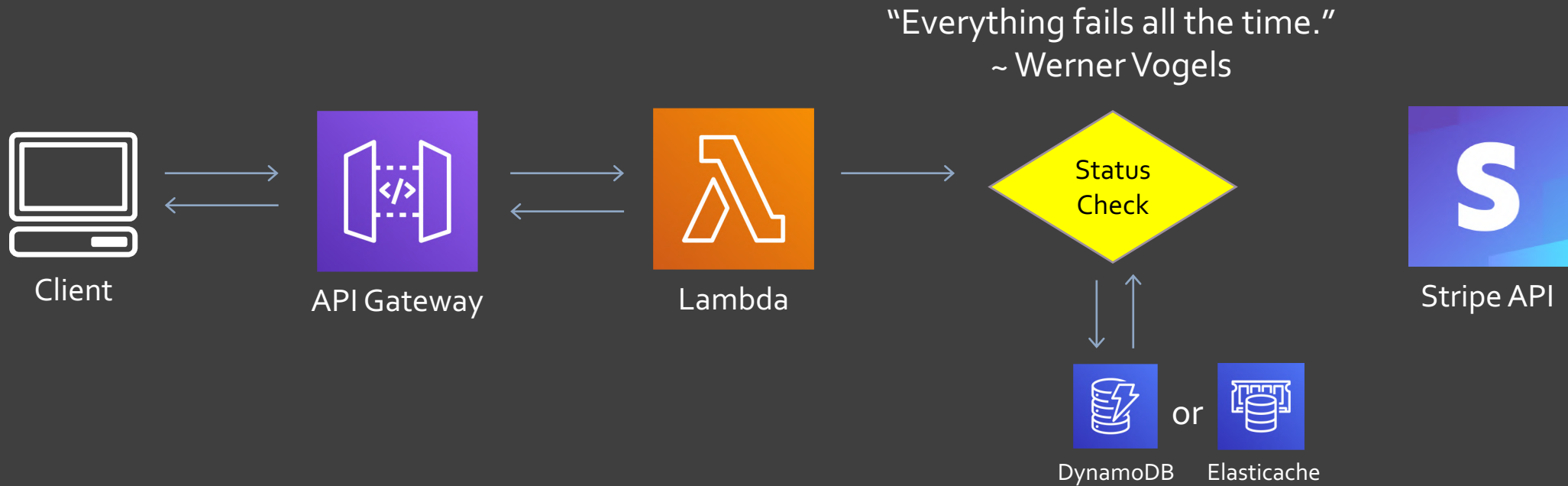
"Everything fails all the time."
~ Werner Vogels



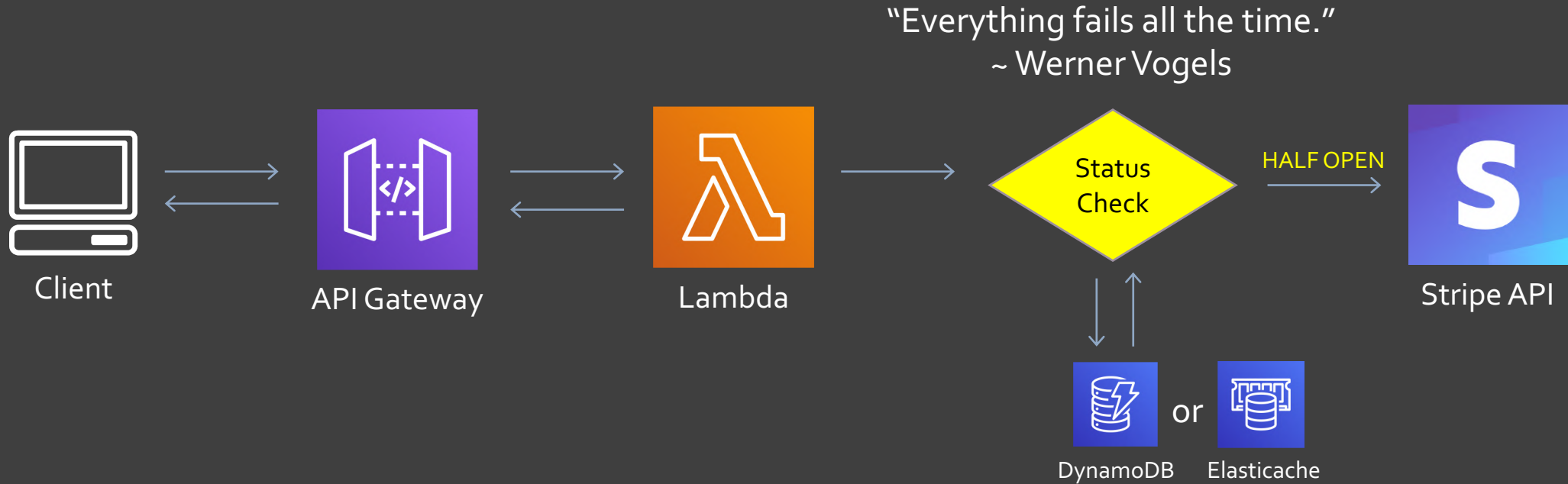
The Circuit Breaker



The Circuit Breaker

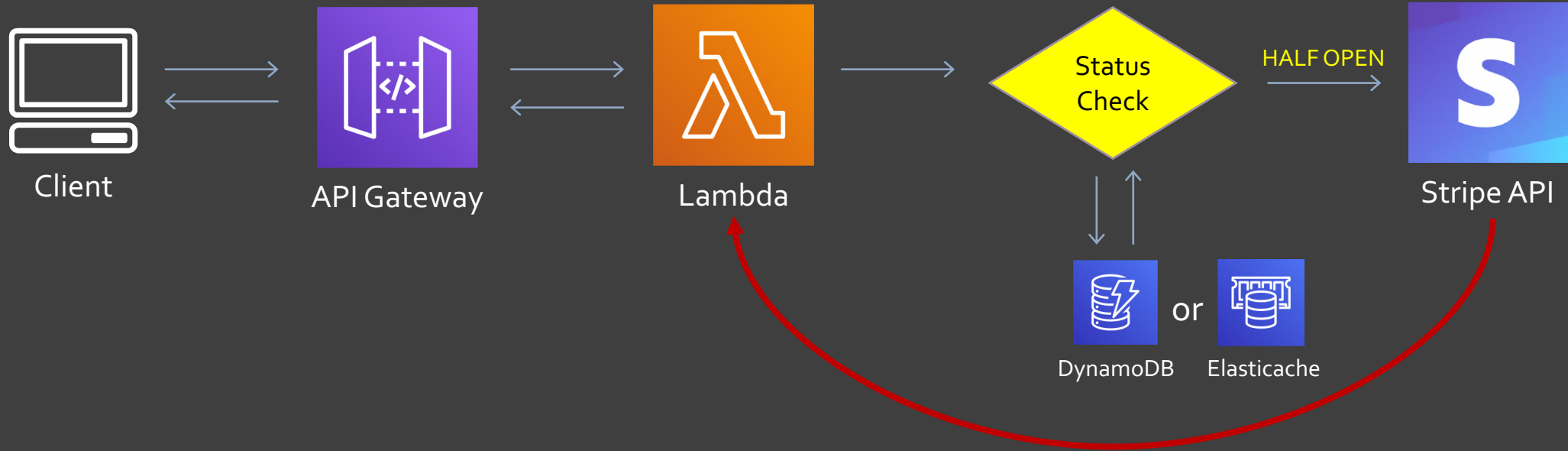


The Circuit Breaker

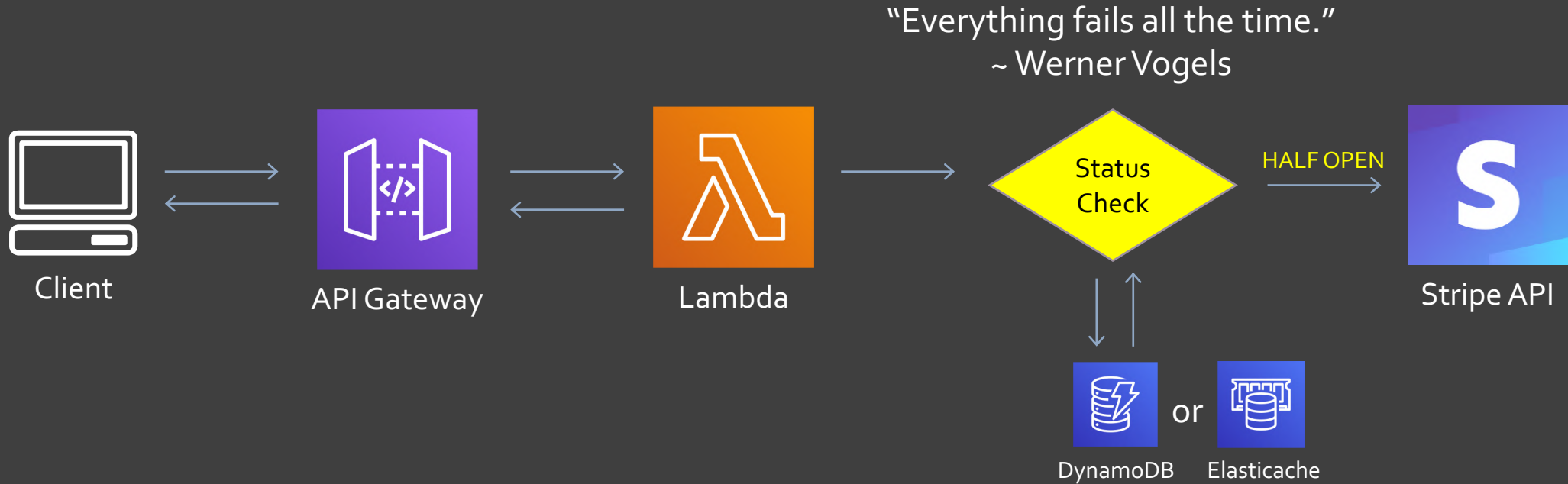


The Circuit Breaker

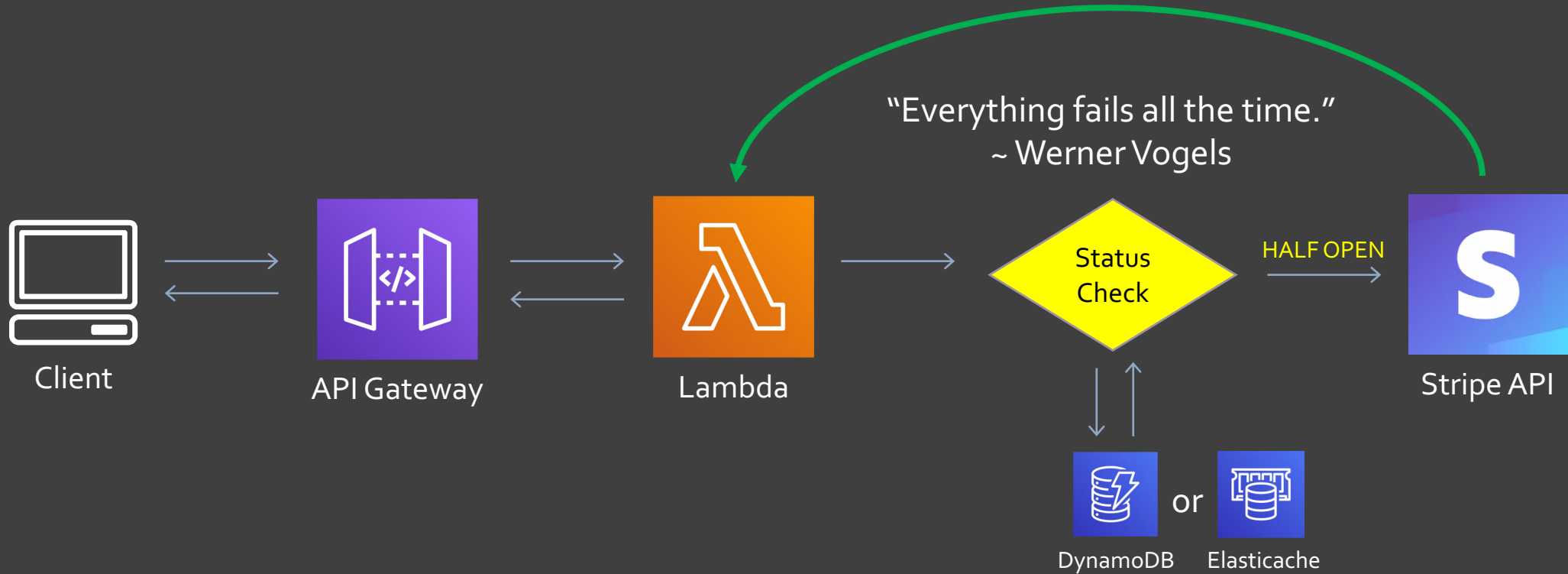
"Everything fails all the time."
~ Werner Vogels



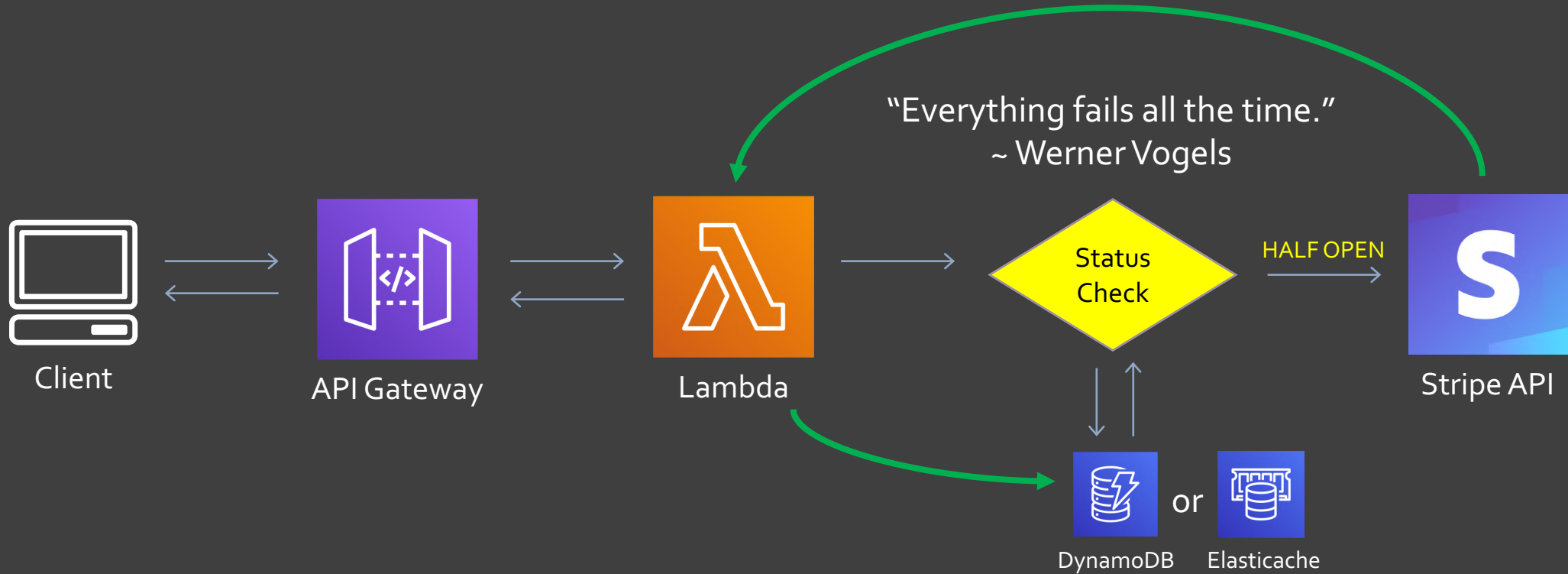
The Circuit Breaker



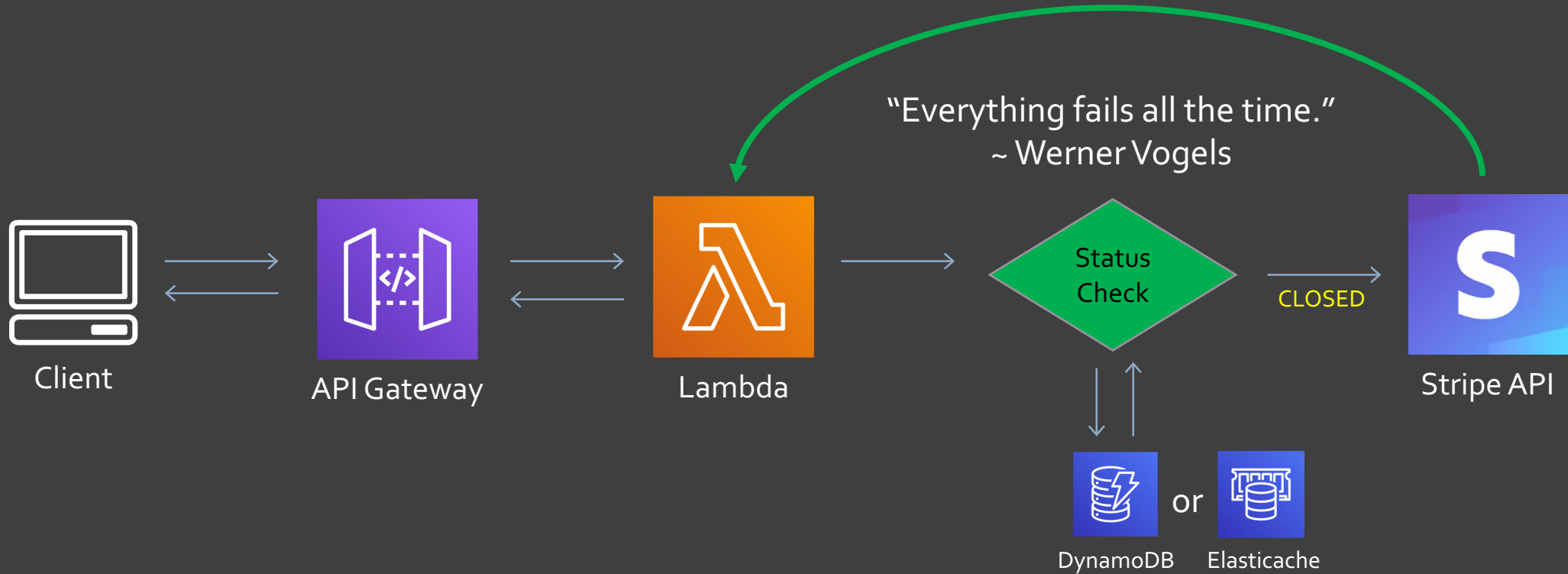
The Circuit Breaker



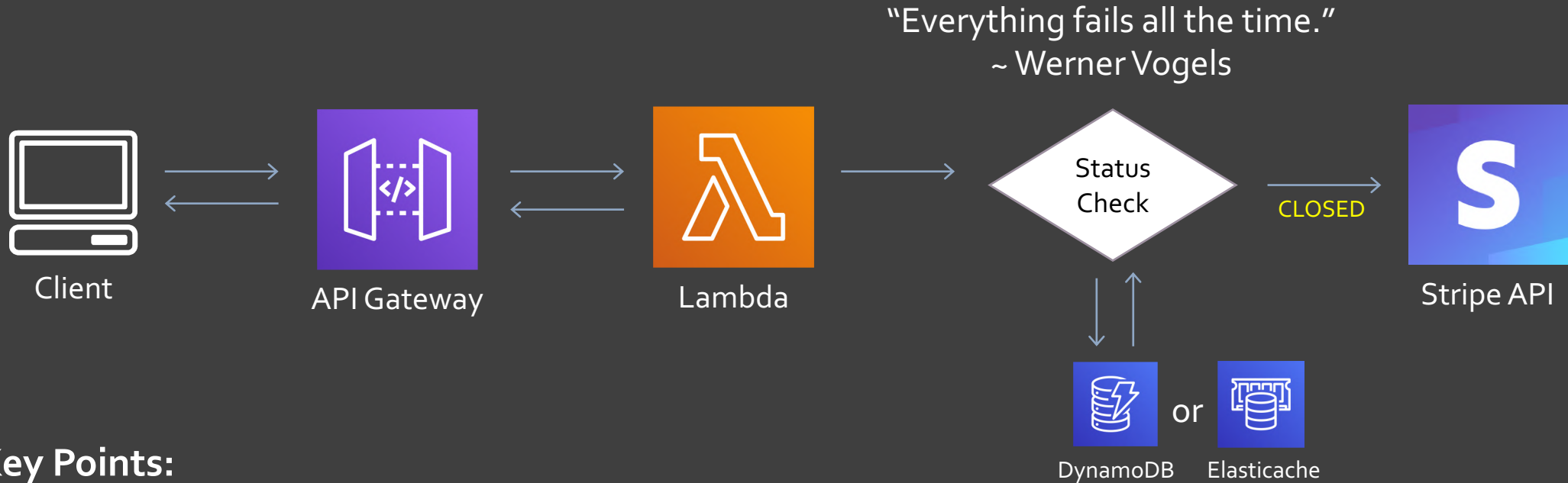
The Circuit Breaker



The Circuit Breaker



The Circuit Breaker

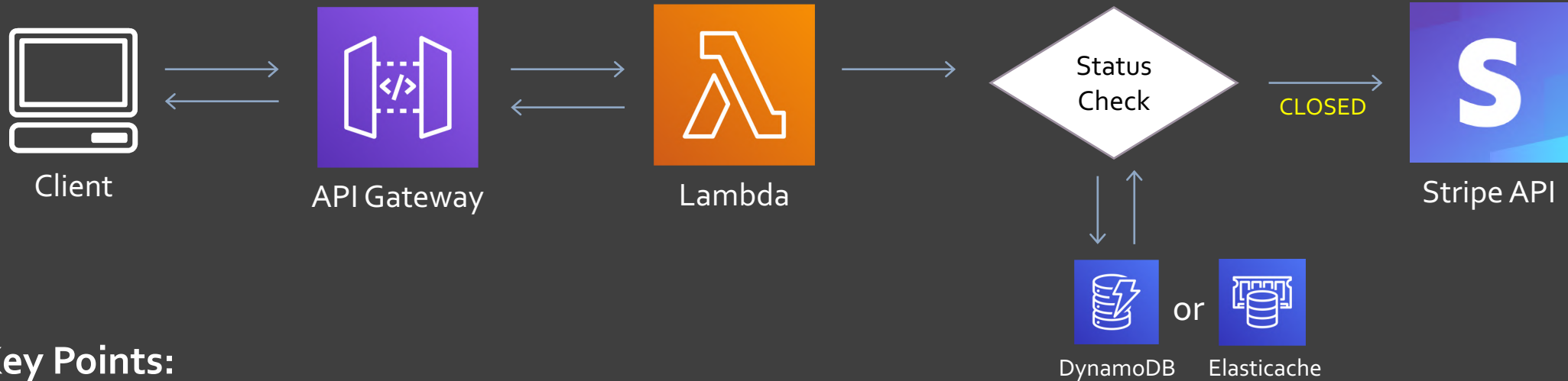


Key Points:

- Cache your cache with warm functions

The Circuit Breaker

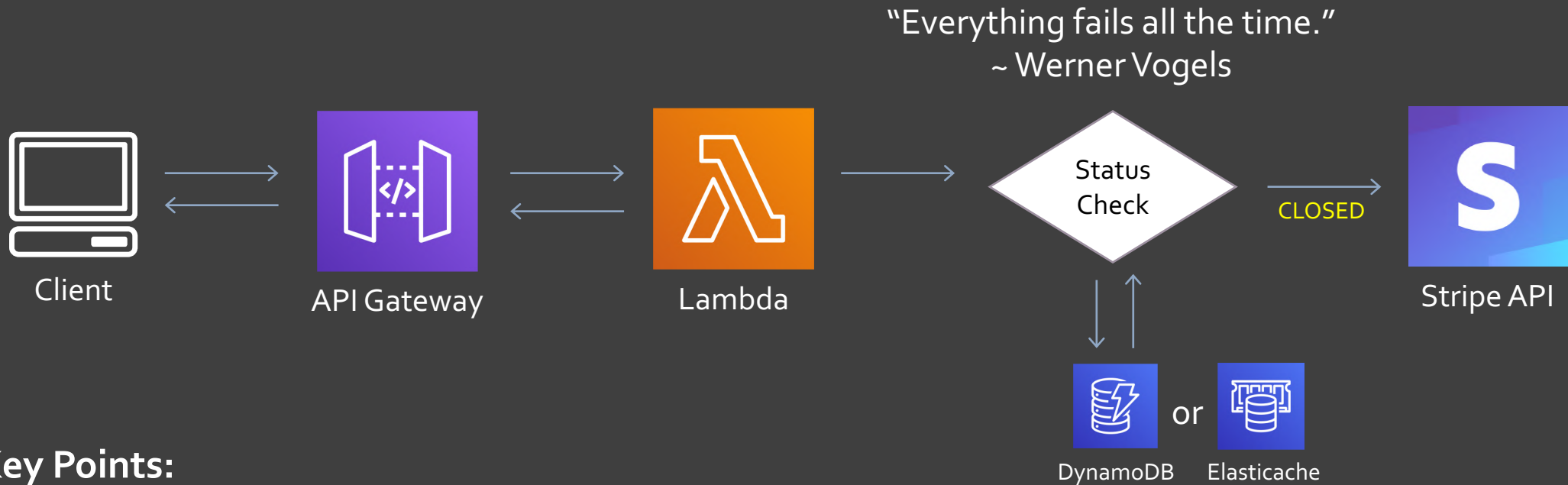
"Everything fails all the time."
~ Werner Vogels



Key Points:

- Cache your cache with warm functions
- Use a reasonable failure count

The Circuit Breaker



Key Points:

- Cache your cache with warm functions
- Use a reasonable failure count
- Understand idempotency!

Key Takeaways

Key Takeaways

- Be prepared for failure – **everything fails all the time!**

Key Takeaways

- Be prepared for failure – **everything fails all the time!**
- Utilize the built in **retry mechanisms** of the cloud

Key Takeaways

- Be prepared for failure – **everything fails all the time!**
- Utilize the built in **retry mechanisms** of the cloud
- Understand **failure modes** to protect against data loss

Key Takeaways

- Be prepared for failure – **everything fails all the time!**
- Utilize the built in **retry mechanisms** of the cloud
- Understand **failure modes** to protect against data loss
- **Buffer and throttle** events to distributed systems

Key Takeaways

- Be prepared for failure – **everything fails all the time!**
- Utilize the built in **retry mechanisms** of the cloud
- Understand **failure modes** to protect against data loss
- **Buffer and throttle** events to distributed systems
- Embrace **asynchronous** processes to **decouple components**

Thank You!

Blog: JeremyDaly.com

Podcast: ServerlessChats.com

Newsletter: Offbynone.io

DDB Toolbox: DynamoDBToolbox.com

Lambda API: LambdaAPI.com

GitHub: github.com/jeremydaly

Twitter: [@jeremy_daly](https://twitter.com/jeremy_daly)



Thank You!

Blog: JeremyDaly.com

Podcast: ServerlessChats.com

Newsletter: Offbynone.io

DDB Toolbox: DynamoDBToolbox.com

Lambda API: LambdaAPI.com

GitHub: github.com/jeremydaly

Twitter: [@jeremy_daly](https://twitter.com/jeremy_daly)



off-by-one
A Weekly Serverless Newsletter



Lambda API

