Recent posts

How you should think about DynamoDB costs

Event-Driven Architectures vs. Event-Based Compute in Serverless Applications

Why I (Still) Like the Serverless Framework over the CDK

Key Takeaways from the DynamoDB Paper

Understanding Eventual Consistency in DynamoDB

Get the DynamoDB Book

Event-Driven Architectures vs. Event-Based Compute in Serverless Applications

February 20, 2023 · 16 min read

Alex DeBrie

Founder, DeBrie Advisory

I recently delivered serverless training to some engineers, and there was confusion between two concepts that come up in discussons of serverless architectures.

On one hand, I describe AWS Lambda as *event-based compute*, which has significant implications for how you write the code and design the architecture in your serverless applications.

On the other hand, many serverless applications use an *event-driven architecture* that relies on decoupled, asynchronous processing of events across your application.

These two concepts -- event-driven architectures and event-based compute -- sound similar and are often used together in serverless applications on AWS, but they're not the same thing. Further, the patterns you use for one will not necessarily apply if you're not using the other.

> KEY POINT
>
> **The event-based nature of Lambda compute is what fundamentally distinguishes Lambda from other computing paradigms and is what leads to the unique constraints and demands of serverless application developers.**

In this post, we'll look at both event-driven architecture and event-based compute. We'll examine the key characteristics of each as well as the implications for you in your applications.

What is an event-driven architecture

Event-driven architectures are all the rage, so we'll start there.

Event-driven architectures are characterized by services that communicate *(1) asynchronously (2) via events*. These two elements distinguish them from other architecture patterns.

Event-driven architectures communicate asynchronously

If you've worked with a frontend client calling a backend GraphQL API or a backend service calling another service via a REST API or RPC, you've used the request-response pattern. This is a common pattern for communication from one client to a service. It is a synchronous flow where the client will wait for a full response from the service indicating the result of the request. A synchronous flow is simple as you get direct feedback on what happened from your request.

However, relying on synchronous communiation has costs as well. It can reduce the overall performance of your application, particularly when working with slower tasks like sending emails or generating reports. Further, the availability of your services goes down. If Service A relies on synchronous responses from Service B to operate, the uptime for Service A cannot be higher than that of Service B. Service B's downtime becomes Service A's downtime.

With asynchronous patterns, these problems are reduced. Services can still communicate, but there's no expectation of an immediate response. The downstream service can process the communication on its own schedule without tying up the upstream service. This adds its own challenges around debugging, eventual consistency, and more, but it does reduce the downsides of synchronous communication.

> For more on request-response vs. event-driven, check out Former AWS Developer Advocate Talia Nassi's post on the benefits of moving to event-driven architectures.

Event-driven architectures communicate via events

Unsurprisingly, the "events" bit is the more unique part of event-driven architectures. In event-driven architectures, services will broadcast events that will be consumed by and reacted upon by other services.

Thus, events require two types of parties:

- **Event producers**, which publish events describing something that happened within the service (a User was created, an Order was delivered, or a Login Attempt failed).
- **Event consumers**, which subscribe to published events and react to them (updating local state, incrementing aggregates, triggering workflows, etc.).

A key feature of true event-driven architectures is that the producers and consumers are completely decoupled -- a producer shouldn't know or care who is consuming its events and how the consumers use those events in their service..

An event producer is like a broadcaster on the nightly news. The broadcaster will say the news that happened regardless of whether anyone is tuned in.

Contrast this with a more traditional message-driven architecture in which one component in a system may insert a message to a message queue (like SQS or RabbitMQ) for processing. A message-driven pattern is also asynchronous, like an event-driven pattern. However, the producer of the message is purposefully sending the work to a specific customer. The use of the queue helps with resiliency and with faster response times from the initial component, but it's not truly "event-driven" in the normal sense of the term given the tight connection between the producer of the message and the consumer.

If event-driven is like a TV broadcast, message-driven is more like your boss sending you an email with a request to create a report. Not only is there a specific output she wants (the report), but there's a specific person she wants to do it (you).

> **NOTE**
>
> Some resources out there will consider message-driven workflows to be a part of an event-driven architecture. I sort of disagree but mostly don't care too much. I do agree that some implications and reasons for using message-driven patterns are similar to event-driven patterns and thus some of the lessons are similar.

The main benefits of event-driven architectures are in their flexibility and resiliency. If I want to add a new consumer of a given event, I don't have to coordinate with the producer of the event. I can start processing the events as they are published and use them in my new service.

Event-driven architectures have been around for a while. If you talk to a developer that spent time at an enterprise shop from the 90s and 2000s, you'll likely hear complaints about the enterprise service bus. More recently, the rise of Apache Kafka and the very effective evangelism from Jay Kreps (one of the original Kafka creators) about the effectiveness of logs & streams has breathed new life into event-driven architectures.

There are a lot of flavors of event-driven architectures out there, including purist patterns like event sourcing and CQRS. I generally recommend avoiding those -- they're fun to think about and imagine all the possibilities, but I've seen them turn into a maintenance and debugging headache.

This was only a cursory review of event-driven architectures. If you want more on this, AWS Developer Advocate David Boyne is the person to follow on all things event-driven. Lots of great stuff on event-driven architectures, including a great set of visuals on Serverless Land.

What is event-based compute

Now that we understand a bit about event-driven architectures, let's turn to event-based compute to see how it differs.

There are two key characteristics of event-based compute:

- First, **the existence of a compute instance is intimately tied to the occurence of an event to be processed**.
- Second, **the compute acts on a single event at a time**.

That's sort of abstract, so let's make it more concrete.

Imagine you create a Lambda function. You write a bit of code, create a ZIP file, upload it to AWS, and set up the configuration in the Lambda service. *Configuring this doesn't actually start your code*, like it might with an EC2 instance or a Fargate container. By default, there will be no actual instances of your Lambda compute running. Your Lambda function has potential, but it hasn't realized it yet.

To make your function a reality, you need to hook it up to an event source. There are a ton of services that integrate with Lambda. The most popular sources are probably API Gateway (HTTP requests), SQS (queue processing), EventBridge (event bus), and Kinesis / DynamoDB Streams (stream processing).

> **WARNING**
>
> Note that, on the Lambda docs page linked above, it uses the term 'event-driven' for a lot of event sources, including many that I would not describe as event driven! More on this below.

Once you've configured your event source and an event has flowed through the configured service, then your function will spring to life. The Lambda service will create an instance of your function and pass the triggering event to it for processing by your function. Your function will process the event as desired and return a response back to the event trigger.

For optimization purposes, the Lambda service may keep your function instance running for a bit to serve other events that occur in a short time period. However, the specifics of this is (mostly) out of your control.

Importantly, the purpose of an instance of running compute is to handle a single, specific event. This distinguishes Lambda from traditional instances or containers which are created to be available to handle requests as they arrive or to poll for messages from a queue or stream. It even distinguishes Lambda from something like creating a Fargate task on a schedule. While the creation of the task is based on an event, the task doesn't naturally have awareness of the event that created it while executing.

Implications of event-based compute

We now know how AWS Lambda is event-based computing -- so what? How does this actually affect our applications?

In my mind, this is the most crucial shift about AWS Lambda. It leads to some of the following preferences from engineers using Lambda:

- Avoiding heavier, existing web frameworks like Django, Express, or Spring Boot in their functions (note that *some* of this is reduced by things like SnapStart)
- Choosing smaller supporting libraries (e.g. Kysely over Prisma)
- Opting for a database that doesn't have connection limits (e.g. DynamoDB) or that manages connection pooling outside of compute (PgBouncer or Amazon RDS Proxy)
- Offloading wait time to the platform (SQS, Step Functions, cron jobs) rather than `setTimeout()` in your application code

Note that none of these preferences is universal -- there's a lot of diversity across Lambda-based architectures. However, they are preferences that you tend to see, particularly in comparison to architectures that use other methods of compute.

There are two main implications from Lambda's event-based nature that you should keep front of mind.

First, **you need to think about statelessness and rapid initialization more than usual**. In an instance-based or container-based application, you can initialize your application, establish database connections, build local caches, and perform other initialization work before making your application available to handle requests.

This is not the case with Lambda compute. Remember, you may be initializing your compute in response to an event. There might be an active HTTP request with a real live user on the other end of it, waiting to see the latest Tweets or to buy some Taylor Swift tickets. You need to make sure your code can load and execute quickly without a bunch of setup.

Specifically, this means keeping your function code small and shallow. Try to avoid loading and initializing nested dependencies across many files. Depending on your tolerance for punishment, you can look at using esbuild or other tools to bundle your code into a single file and reduce init-time disk reads.

Also, consider tips to improve performance on subsequent requests. There are kinds of work that you must do on your cold-start initialization, like establishing network connections to databases or other resources or retrieving dynamic config. However, you should understand how to reuse those resources across multiple requests to avoid making every request to your compute perform like a cold start.

Second, **scaling is about more instances of compute than concurrency within a single instance.** Within a Lambda function instance, you will be processing one and only one event at a time. If multiple events occur at the same time, the Lambda service will spin up more function instances to handle the events. However, it will always be a single event at a time.

This will change not only how you write your application code but potentially even the language you choose. The inimitable Ben Kehoe once said that Node is the wrong runtime for serverless *because* of its async-first approach. This approach is helpful for a traditional backend application where you can handle multiple web requests or operate on batches of queue messages in parallel. However, because Lambda works on a single event at a time, you usually aren't doing a lot of asynchronous work. You're processing a single event in order and thus can benefit from a more straightforward procedural style.

Ben's post is nearly 6 years old at this point, and some of the changes in Node.js since then (specifically async / await) help to sand off the rough edges of Node's model in a Lambda world. But regardless of the language you choose, you should consider how the single-event model affects your application.

Because Lambda scales horizontally across function instances rather than vertically within function instances, you can't share resources across concurrent requests within your compute layer. The easy example here is a pool of database connections, which a traditional web server would use to share across many requests. This leads to the preference noted above where Lambda users prefer databases without connection limits (like DynamoDB) or to implement database pooling outside of their compute (such as via Amazon RDS Proxy).

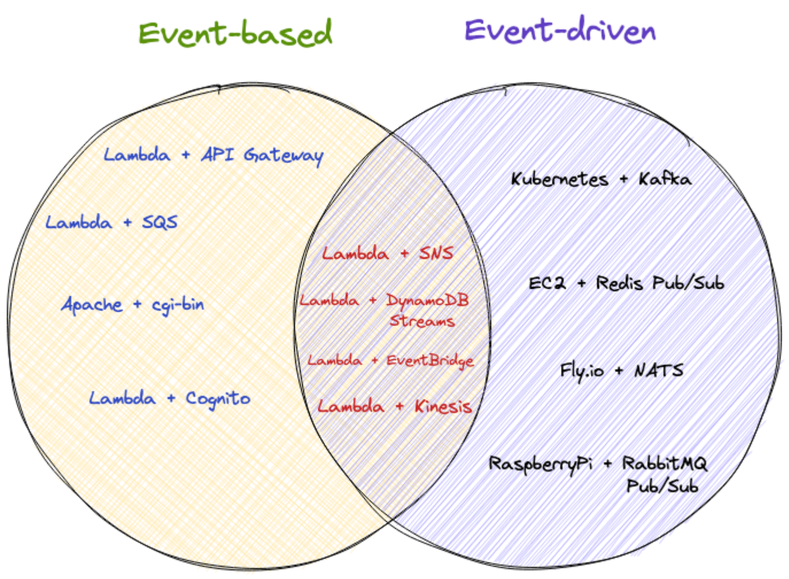
Further, because your compute is working on a single event at a time and is billed as long as it is active, you want to limit the time your application is doing nothing. No more using `setTimeout()` in your web server to handle a background job. More commonly, you'll want to avoid regular long-polling within your function. If you're waiting on something to be done before moving forward, see if you can turn it into an event itself. If you can't do that, try to implement the polling trigger outside of your compute.

Confusion over "event"

Now we know about event-driven architecture and event-based compute, how do they interact? Do you have to use event-driven architectures with AWS Lambda? Can you use event-based compute with Kubernetes?

As mentioned above, I think the confusion in this area is not helped by the Lambda documentation that describes services to interact with Lambda. That document describes a number of sources as "event-driven" that do not fall under the traditional definition of event-driven, such as HTTP requests from API Gateway. Further, integration with most of the services that do utilize event-driven patterns are *not* defined as event-driven, such as DynamoDB Streams, Kinesis Streams, and Apache Kafka.

Luckily for you, I've made a handy Venn diagram to show how these patterns overlap.



In general, the rules are:

- AWS Lambda functions are *always* event-based compute
- You *may* use event-driven patterns with your AWS Lambda functions
- You can use event-driven or non-event-driven patterns with compute other than AWS Lambda

I'm not trying to cover all the cases here. There are other tools (OpenFaaS, Knative) that allow for event-based compute. I'm less familiar with them, but many of the same principles are likely to apply.

Lambda + AWS services: which are event-driven and why?

After publishing this post, Emily Shea had a great suggestion to elaborate on why certain AWS services are or are not event-driven when connected with Lambda. Below is a quick overview. There are caveats here, but I try to outline the general shape of how a service is used with Lambda.

- **API Gateway + Lambda**: Generally not event-driven. This is a request-response pattern as the client will receive a response from your Lambda function indicating the result of the request. If you're using API Gateway in a traditional REST API sense (e.g. "Get User", "Add to Cart", etc), it's not event-driven. It is synchronous and coupled.

  That said, you can use API Gateway as an entrypoint to an event-driven system. For example, a service or a frontend client could publish an event to an API Gateway endpoint, which would send it through to a Kinesis Stream or EventBridge bus. The big tell here will be the response code or body. If you get a response code of `202 Accepted` or a response body that only includes an eventId or messageId property, it could indicate the API is a frontend to an event-driven system.

- **SQS + Lambda**: Generally not event-driven. It is asynchronous, which differs from the synchronous request-response pattern above. However, a queue-based pattern usually has a specific task that the producer desires (e.g. "Send a welcome email after user creation."). Because of this, it's a "message-driven" pattern. May also be called "point-to-point".

  There are caveats here as well. SQS might be used as a buffering / throttling mechanism as part of an event-driven system. For example, a consumer to SNS or EventBridge (discussed below) may push messages to SQS and process from there. SQS can provide greater control and visibility over throttling and retry logic.

- **SNS + Lambda**: Event-driven! 🎉 At least in theory. An SNS topic can have many consumers, and a publisher to an SNS topic usually doesn't have a specific outcome in mind. It is an asynchronous, decoupled pattern.

- **EventBridge + Lambda**: Also event-driven! Basically the same points as SNS here. In practice, EventBridge is probably even more likely to be event-driven than SNS given its singular focus on an event bus with many different event types.

- **Kinesis + Lambda**: Often event-driven! Similar to SNS and EventBridge, but a stream-based solution like Kinesis allows for batch processing instead of individual events as a time. My hunch is that Kinesis is somewhat less likely to be used for 'pure' event-driven architectures as the asynchronous batch processing it enables is good for situations even when the producer and consumer know about each other.

- **Step Functions + Lambda**: Not event-driven! A Step Function state machine executes a specific, defined, multi-step workflow. It may have some asynchronous elements, but there's no de-coupling of producers and consumers here! That said, you may want to introduce some event publishing in between steps of your state machine execution. Yan Cui's post on Choreography vs. Orchestration covers this topic and is one of my favorite serverless posts.

Thanks to Emily Shea for the suggestion and to Maik Wiesmüller for notes on the original writeup. 🙏

Conclusion

In this post, we reviewed the difference between event-driven architectures and event-based compute. We saw that architectures with all types of compute can benefit from event-driven architectures (in the right situations!). Further we saw that all use of Lambda is relying on event-based compute. We also saw some of the implications of event-based compute and the ways in which it affects the design of your application.

If you have any questions or corrections for this post, please leave a note below or email me directly!

Tags: serverless AWS

ALSO ON **ALEXDEBRIE**

Everything you ...
3 years ago · 19 comments
Understanding DynamoDB partitions will make you a better user of ...

A Detailed ...
5 years ago · 31 comments
Look inside the black box of AWS API Gateway to understand authorization, ...

The What, Why, ...
4 years ago · 74 comments
AWS recommends using just a single DynamoDB table for your entire ...

A Guide to S
5 years ago · 2
This post cor overview and S3 Batch ...