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

August 23, 2022 · 21 min read

Alex DeBrie
Founder, DeBrie Advisory

Over the past year or two, I've seen the AWS CDK turn a lot of my friends into converts. And in the very recent past, a few of these people have written up their (mostly positive) experiences with the CDK. See Maciej Radzikowski on why he stopped being a CDK skeptic here or Corey Quinn's list of the CDK's hard edges which, ultimately, is still favorable. For a more skeptical view of the CDK, check out Mike Roberts' excellent thoughts here.

Some of these articles describe similar concerns I have, but none of them quite nails my thoughts on the matter. I thought I'd throw my hat in the ring and describe why I still prefer the Serverless Framework over the CDK.

In this post, I'll start off with my biases and background to help frame my thinking. Next, I'll discuss four reasons why I think the Serverless Framework is a better abstraction for building serverless applications on AWS. Finally, I'll describe some areas that the CDK does do well.

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

My Background and Preferences

Let's start with some facts that factor into my preferences. Your situation might not overlap with mine and, thus, some of my points may not be applicable to you.

First and foremost, I worked for Serverless, Inc., creators of the Serverless Framework, for about two and a half years. This is relevant! I learned a ton about AWS + serverless while working there, and I built a lot of knowledge around the Serverless Framework in particular. I don't have any financial interest in Serverless, Inc., but I do have friends that I like and respect there. In this case, familiarity breeds affinity.

Second, I am primarily building "little-s" serverless applications on AWS. The term "serverless" has gotten pretty slippery now that the marketers have descended, but I generally take this to mean "applications which use primarily managed services glued together with AWS Lambda functions." Yes, it's true that Lambda != serverless, or that you can do "functionless" in your serverless application. I'm more cautious around functionless, but if that's what you're about, have at it! That said, the vast majority of serverless applications on AWS include Lambda at the core.

More importantly for the discussion below is what my applications *don't* have. I very rarely have a VPC, which requires a lot of complicated, boilerplate-y configuration that is easy to screw up if you don't know what you're doing. I don't have security groups, or autoscaling pools, or ECS service definitions that might make using the CDK more pressing. I also don't have a need to create a lot of resources that are similar but slightly different, another area where the CDK could help.

Finally, I have a pretty strong bias for boring in my programming. This claim might seem a little rich for some people, given that I advocate using AWS Lambda and DynamoDB for most applications. You might think something ought to be more commonplace before we call it boring. But I'd actually argue, in solidarity with Brian LeRoux, that Lambda and DynamoDB are fairly boring at this point. They may be newer than some technologies, but it's actually easier to wrap your head around their operating model and, more importantly, their failure modes than those of other technologies.

Another aspect of this boring bias is my preference against cleverness. My first programming language was Python, where explicit over implicit is written into the language itself. It took me a while to learn this lesson, but now I dislike too much indirection in programming. Be kind to your future self -- keep it simple.

These are my biases, and they may be very different from your own. If that's the case, feel free to take the rest of this post with a grain of salt.

Why I like the Serverless Framework

With my own biases stated clearly, let's move into why I like the Serverless Framework. At a high level, there are four reasons I prefer the Serverless Framework:

- It provides a standard structure for every application
- It abstracts the right things
- It doesn't abstract the wrong things
- It constrains our bad impulses

Let's review these in detail.

A standard structure for every application

We're already a few hundred words into this post, and I haven't even described how the Serverless Framework and the AWS CDK differ.

Both tools are basically pre-processors for AWS CloudFormation. Their core difference is in the process they use to get to CloudFormation.

With the Serverless Framework, you're writing a single, declarative YAML configuration file called `serverless.yml` that will describe your application. It will look something like the following:

```yaml
service: users-service

frameworkVersion: '3'

provider:
  name: aws
  runtime: nodejs14.x
  environment:
    TABLE_NAME: !Ref DynamoDBTable
  iam:
    role:
      statements:
        - Effect: 'Allow'
          Action:
            - 'dynamodb:GetItem'
            - 'dynamodb:PutItem'
          Resource: !GetAtt 'DynamoDBTable.Arn'

functions:
  createUser:
    handler: src/handlers/createUser.handler
    events:
      - http: POST /users
  getUser:
    handler: src/handlers/getUser.handler
    events:
      - http: GET /users/{userId}

resources:
  Resources:
    DynamoDBTable:
      Type: "AWS::DynamoDB::Table"
      Properties:
        ...
```

It has a number of top-level blocks, including `functions` for creating Lambda functions and `resources` for provisioning other AWS resources.

The key abstraction is really in the `functions` block, as it focuses on simplifying the process around creating Lambda functions and hooking up event sources to these functions. This is the core of most serverless applications, and simplifying this process is a big win. We'll talk about this further in the next section.

The CDK takes a different approach. Rather than a declarative configuration file, the CDK has you write imperative code in a general-purpose programming language (TypeScript, Python, Golang, etc.) to describe the infrastructure you want. With this structure, you can use things like functions to abstract common operations or loops to create multiple resources with similar configuration. Further, you can have abstracted modules from other files or even other NPM packages that encapsulate resource creation logic.

I like the Serverless Framework because I always know where I am. If I come to an existing project that I've never seen before, I can look for the `serverless.yml` file and understand the structure of the application -- what functions there are, where they're located in the repository, how they're triggered, and what supporting resources they use.

If you're from the Rails ecosystem, you might think of the Serverless Framework's approach as 'convention over configuration'. Mohamed Said recently made the same point about Laravel applications:

> The biggest reason why following the Laravel convention is so important is that I can look into a team's work after weeks and I know exactly where to look to find

> things.
>
> — Mohamed Said (@themsaid) June 24, 2022

In contrast, every CDK repo I've looked at turns into a murder mystery. I'm digging through application code that is mixed with infrastructure code, trying to find my function handlers, my service code, anything that will help me solve The Case of the Missing IAM Permission.

Some people have discussed the long-term issues around CDK maintainability -- what if people aren't updating your constructs or staying on top of CDK version updates?

Those issues are real, but I'm way more worried about how you maintain developer knowledge around a specific CDK application. There are real hurdles to onboarding a new developer into your custom structure. They don't just need to know the CDK, they need to know *your team's specific implementation* of the CDK.

The knowledge of your application that is built up within your team is hidden, and it will result in slower ramp-up times for new developers.

The Serverless Framework abstracts the right things

I mentioned in the last section that both the Serverless Framework and the CDK are abstractions over CloudFormation. In my opinion, the Serverless Framework does a better job of abstracting the right things to make it easier to work with serverless applications.

There are two hard things about deploying serverless applications:

- Packaging up your functions
- Configuring the events that will trigger your functions

The Serverless Framework has rightly identified that functions and events are core to serverless applications and has made them the core abstraction. Let's look at another example `functions` block in a `serverless.yml` file:

```
functions:
  createUser:
    handler: src/handlers/createUser.handler
    events:
      - http: POST /users
  processQueue:
    handler: src/handlers/processQueue.handler
    events:
      - sqs:
          arn: !GetAtt [MyQueue, Arn]
          batchSize: 10
  fanout:
    handler: src/handlers/fanout.handler
    events:
      - sns:
          arn: !Ref MyTopic
          topicName: MyTopic
```

When I run `serverless deploy` to deploy my stack, it's going to build and configure a function on AWS for each function in my `functions` block (here, there are three: `createUser`, `processQueue`, and `fanout`). For each function, it may install your dependencies, run a build process, zip up the directory, upload it to S3, and register the function and its basic configuration with the Lambda service. You don't really need to know much about that packaging process, unless you have specific needs.

To be fair to the CDK, some of their function abstractions, like the NodeJsFunction construct do *most* of this too. Anecdotally from friends, I've heard that the function build part of CDK is under-loved compared to the core infrastructure deployment part, but it's functional.

One example of this is from Roman here:

> That’s good defaults I think. How’s your overall experience with these lambda constructs? I find it still problematic to build, test and publish ts lambdas
>
> — Roman (@naumenko_roman) August 23, 2022

The bigger difference between the two is in the second hard part of serverless applications -- configuring events to trigger your functions.

In the `serverless.yml` example above, there's a consistent format for potential trigger. Each function has an `events` property where you can configure an array of events (though it's usually one per function). Each configured event has a type and any required properties for the event.

Thus, configuring a Lambda function to connect to API Gateway to handle HTTP requests feels very similar to configuring a function to consume from an SQS queue or to handle SNS notifications.

And sometimes these configurations are quite complex! Creating a Lambda function to handle an API Gateway endpoint requires creating 6-8 resources, including two sub-resources on an API Gateway instance (a Resource and a Method), an IAM role for your function, plus a resource-based permission on your function so API Gateway can invoke it.

That's a lot! With the Serverless Framework, that's abstracted away from you. Further, it's not abstracted in a way that harms you (as we'll discuss in the next section).

If you compare configuring these three events types in the CDK, you'll see there are three completely different ways to manage it.

With API Gateway, you need to create the API Gateway REST API instance. Then, you need to create a resource on the API (or multiple resources, if this is a nested route). Finally, you need to add a method on your created resource that integrates with your created Lambda function. Check an example of this from Borislav Hadzhiev here.

If you're doing an SQS integration, the format is completely different. Under the hood, certain Lambda integrations use event source mappings, and the CDK exposes that directly to you. Thus, connecting a Lambda function to an SQS queue looks like this:

```
declare const fn: lambda.Function;
const queue = new sqs.Queue(this, 'MyQueue');
const eventSource = new SqsEventSource(queue);
fn.addEventSource(eventSource);
```

Source: CDK docs

Notice that you create a Lambda function, create an SQS Queue, and then create a third thing to connect them together. Notice that this third thing, the event source mapping, is attached to the Lambda function.

If we go to the third type of event source, connecting to an SNS topic, we find a third pattern:

```
const myTopic = new sns.Topic(this, 'myTopic');
declare const myFunction: lambda.Function;
myTopic.addSubscription(new
subscriptions.LambdaSubscription(myFunction));
```

Source: CDK docs

Similar to SQS, we create a Lambda function and the related resource (an SNS topic). However, to connect them, we *attach the function to the resource* (rather than the resource to the function, like in SQS).

In each of these three examples, CDK more accurately reflects the resources required to create and configure these integrations. However, I think it retains some complexity that can easily be abstracted away. For the most part, I don't need to know about event source mappings vs. API Gateway Lambda integrations (though I do need to know about the various types of error handling across different services!).

I think this leads to a confusing mental model of how serverless applications on AWS should be built. You should think of it as a Lambda-centric model (or perhaps "compute-centric", if you're a functionless fanatic) where you're using lots of managed services but tying them together via Lambda. With the Serverless Framework, Lambda functions and their events are the core, and the supplemental resources are the leaf nodes. With CDK, it puts all resources on the same footing, making it harder to organize how your application works.

The CDK abstracts the wrong things

On the flip side of the previous section, I also think that the CDK abstracts the wrong things.

But first, I have a confession. In the previous section, I made the Serverless Framework event configuration look slightly easier than it is. For both the SQS and SNS integrations, we're referring to additional infrastructure in our application -- an SQS queue and an SNS topic.

While the Serverless Framework looks at functions and events as the core of your serverless applications, it also realizes you will need additional supporting infrastructure to do anything meaningful. That's where the `resources` block of your `serverless.yml` comes in.

In the `resources` block, you can configure any additional AWS resource you want via straight CloudFormation. For our SQS- and SNS-enabled application above, our `resources` block would look as follows:

```
resources:
  Resources:
    MyQueue:
      Type: AWS::SQS::Queue
      Properties:
        VisibilityTimeout: 60
    MyTopic:
      Type: AWS::SNS::Topic
      Properties:
        TopicName: MyTopic
```

The part I love about this is that you're actually writing CloudFormation. You're learning the tool that you'll be using under the hood.

There are a lot of complaints about CloudFormation, but they mostly center on the verbosity of it -- there's just *so much* I have to write. And it's *boilerplate*!

But as we saw in the last section, most of the boilerplate CloudFormation has been abstracted from you in functions and events. The resources you create in the `resources` block are pretty flat and direct. There's not a lot you can abstract away in SQS queues, SNS topics, or DynamoDB tables. (For the latter, AWS SAM tried with the `SimpleTable` resource, but it turns out the other properties of DynamoDB tables are pretty useful!)

Additionally, your Lambda functions will need to have the requisite IAM permissions to interact with your resources, and you'll need to specify those in IAM-policy format using the `iam` block in `serverless.yml`.

I think part of the attraction of CDK is that it abstracts both raw CloudFormation and IAM. Rather than writing declarative YAML, you're invoking a method with some properties. And rather than detailing the specific IAM statements you need, you're doing something like `myTable.grantReadWriteData(myFunction)` to allow your function to read and write to your DynamoDB table.

But at some point, you need to learn those fundamentals. If your deployment fails, it's going to fail in a CloudFormation-specific way. If your permissions don't work, it's going to tell you the specific resource and IAM action combination that was disallowed. If that is completely foreign to you, you're not going to be able to debug effectively.

AWS is a massive ecosystem -- thousands of services with intricate connections, complex auth requirements, and loads of pitfalls. You can do some amazing things, but it's truly daunting to newcomers.

I was able to massively upskill my AWS ability through the Serverless Framework, and I've seen the same for others. With the Serverless Framework, you can get started with a simple function and event connection quickly, without learning the specifics of CloudFormation and IAM. Then, you can gradually layer in new pieces -- as you want persistent storage, you add a DynamoDB table resource and the needed PutItem and GetItem permissions. It's all incremental, so you're learning as you go, rather than tossing you into the deep end once something inevitably goes wrong.

The CDK enables our bad impulses

The previous section was about how the Serverless Framework helps beginners to AWS. But you, dear reader, may not be a beginner. Maybe you think you don't need the training wheels that the Serverless Framework provides. I'm here to tell you that you may need the structure that the Serverless Framework provides more than anyone.

CDK proponents love to mention that you can write code for your infrastructure. But one of my biggest problems with the CDK is that you can write code for your infrastructure. I don't mean to turn into a jaded old man, but I've seen some stuff in CDK applications. Stuff that would give you nightmares.

As engineers, we can be too clever for our own good. We can fall in love with abstraction, with DRY-ing up our code, or with *beautiful* code. But too often, these elegant edifices are impenetrable to outsiders or difficult to maintain over time.

There are two specific problems I've seen with the CDK, though I'm sure there are others.

The first is that it's too easy to create a mess of resources without understanding the underlying purpose or intent. The most common place I see this is with an inexplicable explosion of CloudFormation Stacks. One CDK application I saw used seven different stacks for a pretty simple service (~8 HTTP endpoints, an S3 bucket, a DynamoDB table, and a few state machine definitions). Note that these weren't CloudFormation Nested Stacks, which are inherently tied to one another, but were separate, independent stacks, which made deployment (and failures in deployment) much more complicated.

The second is in over-emphasizing abstractions that can be reused across teams. This is a core concept in CDK -- there are different levels of 'constructs' from low-level CloudFormation resources ("L1 constructs") to highly specific patterns ("L3 constructs"). People get fired up about this because they think they can write this One Resource to Rule Them All to solve all their problems, but the reality is a lot trickier.

I think the average software developer (and I include myself in this group!) is actually not that great at writing *reusable, library* code. This is doubly true for the average CDK-adjacent developer (again, I'm in this group!) who is either a DevOps-y shaped engineer that's doing some product work or a full-stack-y engineer that's learning just enough AWS to be dangerous. Few developers boast a deep knowledge of cloud infrastructure combined with the ability to create useful abstractions.

Taylor Otwell, creator of Laravel, shared a screenshot of an HN discussion recently that hit on similar points:

> Suspect this is also very true for Laravel projects. 😅 pic.twitter.com/reyAWYkH6U
>
> — taylor otwell 🪶 (@taylorotwell) August 8, 2022

Good, solid abstractions are hard to make. But they're sooo tempting to try. In most cases, you're better off relying on proven, tested abstractions rather than trying to write your own.

What the CDK does well

I've spent a lot of time talking about what I don't love about the CDK, but it can't be all bad. The CDK is getting a lot of adoption in the AWS circles in which I travel, and many people whom I respect are loving the CDK. I do want to outline some of the benefits. And because I refuse to have any fun, I'll also point out the downside of each one.

Given that, what is the CDK doing well that is accounting for its popularity? There are three core areas I see.

First and foremost, the CDK removes some of the drudgery of infrastructure-as-code. As much as I like the Serverless Framework, writing YAML can be boring. Moving from declarative template-land into imperative coding-land can be a lot more fun. Plus, there are some benefits around auto-complete, and the L2 constructs can help with the low-level bits of gluing these things together.

I'm cautious about this benefit, because fun does not necessarily equal good. Further, as mentioned above, I think the Serverless Framework abstracts a lot of the true drugery of little-s serverless applications. However, we have to acknowledge this impetus as a real factor. And if you're a more experienced AWS engineer that understands IAM, CloudFormation, and the IaC process, you're not losing as much by going to the CDK.

A second benefit of the CDK is an easier way to create lots of similar resources. While on the excellent aws.fm podcast, Matthew Bonig noted he had a situation where a team was making hundreds of similar RDS instances. The copy-pasta required to create all of these or to encode best practices can be difficult, and the CDK makes this much easier than a more declarative format.

That said, I've only had this situation come up one time in my ~6 years of doing serverless development, and even that was for a resource that had 6-7 similar instances. The slight copy-pasta required to maintain it wasn't a burden. But if this applies to you, go for it!

A third, more common, benefit is the ability to abstract truly boilerplate stuff. If you're creating a VPC or certain other AWS solutions that require a ton of configured resources, being able to abstract that in a single construct that has sane defaults is a big benefits.

The main downside to this is that you still are responsible for owning that infrastructure! The CDK can make it easier on Day 1, but that infra is still yours on Day 2.

It reminds me of this tweet from Gwen Shapira:

> All AWS EKS "get started" docs tell me to run CloudFormations. I ended up with 32 resources - mostly network related. I have no idea why they all exist and what each does.
>
> How will I debug things if they break? or help someone else if my stuff doesn't work for them?
>
> — Gwen (Chen) Shapira (@gwenshap) August 15, 2022

Gwen is talking about a standard CloudFormation template, but the point is the same. If you're using something off the shelf to avoid learning about it, it's a ticking time bomb as to when you'll actually have to learn it. And at that point, the pressure will be higher.

Finally, **I like the experimentation that the CDK is doing.** Tech and software moves in fits and starts, and it's hard to predict what will work. There have been lots of tech trends that I've been skeptical of or downright hostile to. Sometimes I was wrong!

Infrastructure as code is not a solved problem yet. Let a thousand flowers bloom.

Conclusion

In this post, I described why I still prefer the Serverless Framework for building serverless applications on AWS. I prefer the way it constrains our impulses, its convention over configuration, and the way it incrementally teaches the cloud to users. I also did a quick run through of things the CDK does well.

I realize opinions will differ on this, and that's OK! I'm here to describe the tradeoffs as I see them and in light of my situation and preferences. If yours differ, then your conclusion might as well. Hit me up and let me know why I'm wrong :)

Thanks to Matt Bonig for his comments on this post. Be sure to check out The CDK Book, an excellent resource he wrote with three other smart people. All mistakes are mine.

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**

How and Why to Use CloudFormation ...
5 years ago · 2 comments
Macros are a powerful tool for making CloudFormation easier to work with. Learn ...

DynamoDB Transactions ...
4 years ago · 2 comments
DynamoDB has added support for transactions. In this post, see the ...

SQL, NoSQL, and Scale: How ...
4 years ago · 20 comments
Understand when and why relational databases don't scale, and how ...

The Three I Limits You I
3 years ago · 3
Learn about t important lim DynamoDB a