Provisioning a Virtual Private Cloud at Scale with AWS CDK

by Francois Rouxel | on 08 OCT 2020 | in [Amazon DynamoDB](https://aws.amazon.com/blogs/apn/category/database/amazon-dynamodb/), [Amazon EC2](https://aws.amazon.com/blogs/apn/category/compute/amazon-ec2/), [Amazon VPC](https://aws.amazon.com/blogs/apn/category/compute/amazon-vpc/), [AWS Cloud Development Kit](https://aws.amazon.com/blogs/apn/category/developer-tools/aws-cloud-development-kit/), [AWS CloudFormation](https://aws.amazon.com/blogs/apn/category/management-tools/aws-cloudformation/), [AWS Partner Network](https://aws.amazon.com/blogs/apn/category/aws-partner-network/), [Customer Solutions](https://aws.amazon.com/blogs/apn/category/post-types/customer-solutions/), [Developer Tools](https://aws.amazon.com/blogs/apn/category/developer-tools/), [Expert (400)](https://aws.amazon.com/blogs/apn/category/learning-levels/expert-400/), [Technical How-To](https://aws.amazon.com/blogs/apn/category/post-types/technical-how-to/) | [Permalink](https://aws.amazon.com/blogs/apn/provisioning-a-virtual-private-cloud-at-scale-with-aws-cdk/) | [Comments](https://commenting.awsblogs.com/embed.html?disqus_shortname=aws-apn-blog&disqus_identifier=19883&disqus_title=Provisioning+a+Virtual+Private+Cloud+at+Scale+with+AWS+CDK&disqus_url=https://aws.amazon.com/blogs/apn/provisioning-a-virtual-private-cloud-at-scale-with-aws-cdk/) | [Share](https://aws.amazon.com/blogs/apn/provisioning-a-virtual-private-cloud-at-scale-with-aws-cdk/)

*By Francois Rouxel, Solution Principal and APN Ambassador at Slalom*

[](https://aws.amazon.com/partners/ambassadors)Infrastructure as code (IaC) is one of the most important concepts used with cloud solutions. It provides a way to treat your infrastructure as if it were a coded application. The resulting code is stored in a repository and can be part of any CI/CD pipeline.

Amazon Web Services (AWS) provides a tool to enable infrastructure as code, [AWS CloudFormation](https://aws.amazon.com/cloudformation/), which deploys stacks and provisions your resources on AWS using JSON or YAML files called templates.

In this post, I will show you how to create 100 virtual private clouds (VPCs) in one AWS region without providing any parameters; the default limit is five VPCs per region. More, you will see how to easily establish a peering connection between two of them within a single line of code.

I am a Cloud Solutions Architect at Slalom and have been an [APN Ambassador](https://aws.amazon.com/partners/ambassadors) since early 2020. APN Ambassadors work closely with AWS Solutions Architects to migrate, design, implement, and monitor AWS workloads. I have been working in IT for 20 years and have oriented my interest and expertise in the cloud. My role at Slalom is focused on helping companies migrate to AWS.

[Slalom](https://www.slalom.com/platforms/aws-cloud-consulting) is an [AWS Premier Consulting Partner](https://aws.amazon.com/partners/find/partnerdetails/?n=Slalom&id=001E000000NaBI0IAN) with multiple AWS Competency and Service Delivery designations. Slalom is a modern consulting firm focused on strategy, technology, and business transformation.

**Here Comes the CDK**

Writing an AWS CloudFormation template can be challenging sometimes, as it does not have a dynamic nature. For instance, it’s not possible to create a loop in the template to provision the same infrastructure (subnets and Network Access Control lists, for example) over many [AWS Availability Zones](https://aws.amazon.com/about-aws/global-infrastructure/regions_az/).

The [AWS Cloud Development Kit](https://aws.amazon.com/cdk/) (CDK) is a tool that helps you create and deploy CloudFormation templates. CDK was released in October 2019 and offers valuable features like loops. In this post, you will see how to use some of those features to create and deploy VPCs at scale.

**VPC Peering**

A virtual private cloud (VPC) is an isolated private network in which you can provision resources. It can be peered or connected to another VPC or network on-premises.

Once your VPC is peered with another one, instances from the peered VPC can see all of the instances in your VPC. This, of course, depends on the security groups you may have configured, but if your peering connection is well set in your route tables, the connectivity is here.

**Things to Consider When Creating a VPC**

One main attribute of the VPC is its CIDR range (basically, it’s size). AWS recommends provisioning the biggest VPC you can, which is /16. For instance, you can create a VPC using 10.0.0.0/16, which means you can have up to 65,536 IPs from 10.0.0.0 to 10.0.255.255.

Things get complicated when you want to peer your VPCs with others that have overlapping CIDR ranges. In this case, the peering connection won’t succeed. This post shows you how to create VPCs without having to worry about this detail.

**Solution Overview**

In an IT project, it’s common to see many teams working on different features. A typical pattern is for each team to have a dedicated Git branch created from the main branch of the project. The CI/CD pipeline will build and deploy the solution for each branch.

In a case where there are two development teams, we could have two branches (dev1 and dev2), and the main branch (staging) would be the integration branch, where you will merge the feature of each team.

So, we are going to create three VPCs in the same account, in the same region:

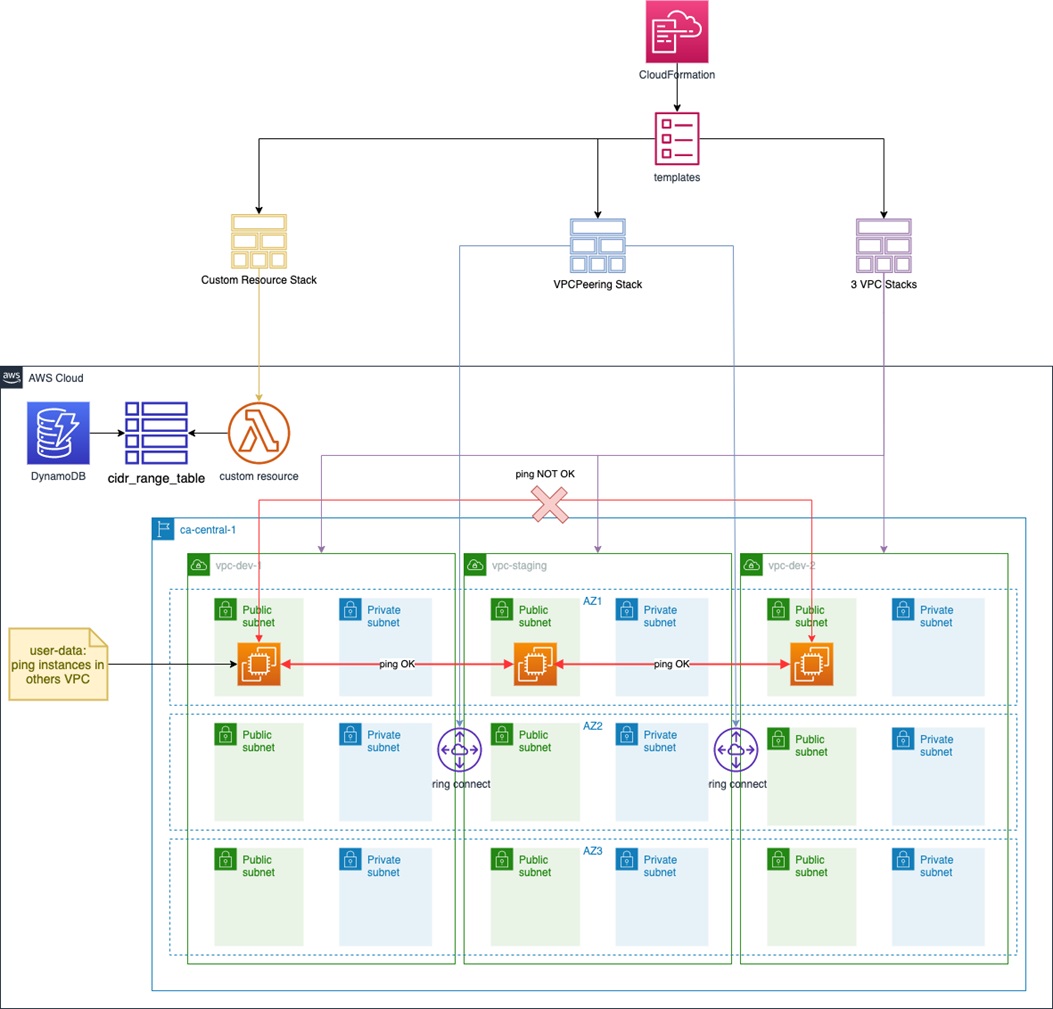
* 2 dev VPCs (vpc[BD3] -dev-1 and vpc-dev-2)
* 1 staging VPC (vpc-staging)
* vpc-dev1 and vpc-dev2 will be peered to vpc-staging, but not between each other

Note that we could have chosen different accounts and regions. As an example, we could have deployed vpc-dev-1 and vpc-dev-2 in ca-central-1 in a development account, and vpc-staging in us-east-1, in a test account. Even with this setup, VPC peering is possible.

A custom resource (such as an [AWS Lambda](https://aws.amazon.com/lambda/) function) is used to know when all stacks are complete. This function will update the [Amazon DynamoDB](https://aws.amazon.com/dynamodb/) table to list all the VPC and subnets deployed during the process.

In the architecture diagram below, you see three stacks created from the CloudFormation templates:

1. One stack for the custom resource to create the Lambda function that’s called when the deployment is complete.
2. One stack to create the VPCs.
3. One stack to peer vpc-dev-1 and vpc-dev-2 with vpc-staging; it means that an instance deployed in vpc-dev-1 (or vpc-dev-2) will be able to communicate with an instance in vpc-staging.

[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-1.jpg)

*Figure 1 – Solution overview diagram.*

Note that in the solution, the private subnets creation has been commented on purpose to get the solution deployed faster. Just uncomment the following lines to provision the private subnets (it will also provision Nat gateways):

subnet\_configuration=[

ec2.SubnetConfiguration(

subnet\_type=ec2.SubnetType.PUBLIC,

name="Ingress",

cidr\_mask=24

#), ec2.SubnetConfiguration(

# cidr\_mask=24,

# name="Application",

# subnet\_type=ec2.SubnetType.PRIVATE

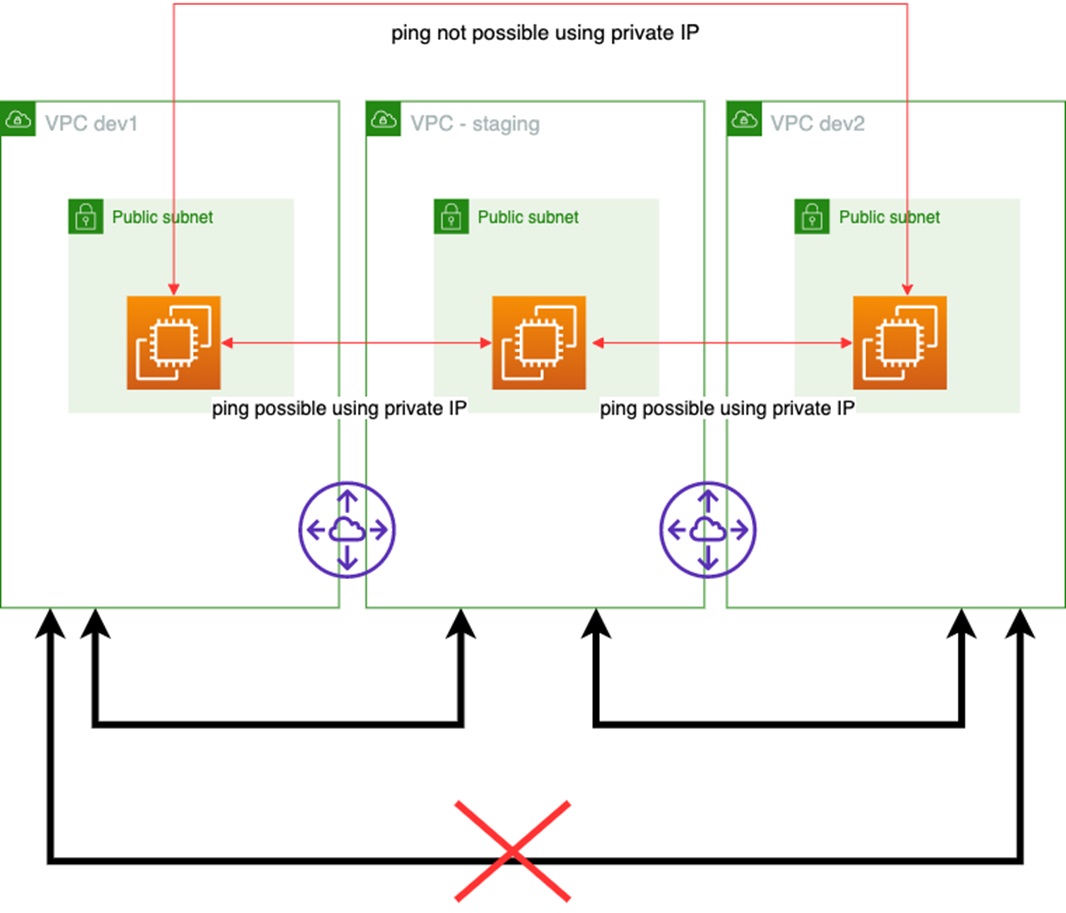
)

]

**VPC Peering Transitivity is Not Allowed**

One important concept with AWS and peered VPCs is the non-transitivity of VPC peering. Take our three VPCs, for instance:

* vpc-dev-1 is peered with vpc-staging, which means connectivity between vpc-dev-1 and vpc-staging is possible.
* vpc-dev-2 is peered with vpc-staging, which means connectivity between vpc-dev-2 and vpc-staging is possible.

[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-2.jpg)

*Figure 2 – Non-transitivity.*

To verify this statement, the solution will deploy an [Amazon Elastic Compute Cloud](https://aws.amazon.com/ec2/) (Amazon EC2) instance in each VPC. Each EC2 instance will try to ping every minute the other instances, using their private IPs, and will output the result into one of our DynamoDB table.

Note that peering two VPCs together is not enough to make the ping possible. The solution makes sure the security group attached to each EC2 instance allows ICMP traffic within the three VPCs, and that each route table has a route that allows the networking connectivity.

**What About CIDR Ranges?**

Amazon DynamoDB is a very powerful NoSQL database and a perfect place to store key/value pairs. I will use it to store the VPC CIDR ranges and their subnets.

Two DynamoDB tables are created using CDK:

* cidr\_range\_table – Stores all the VPC info.
* last\_cidr\_range\_table – Stores the last CIDR range used.

**Getting Started**

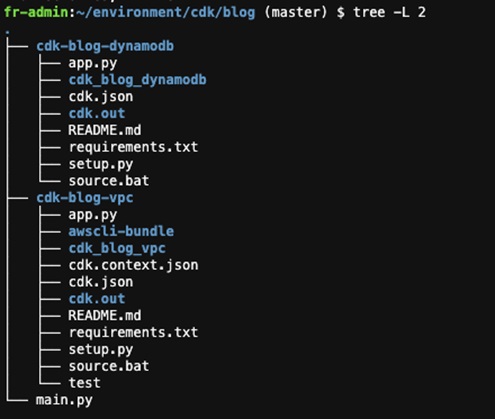
This post is a good opportunity to introduce [AWS Cloud9](https://aws.amazon.com/cloud9/), an Integrated Development Environment (IDE). AWS Cloud9 runs on either Amazon EC2 or your locale machine. One nice feature is that it comes with useful pre-installed tools like Git, Python, [AWS Command Line Interface](https://aws.amazon.com/cli/) (CLI), or CDK.

Here are the steps to use AWS Cloud9:

1. First, you need an [AWS Identity and Access Management](https://aws.amazon.com/iam/) (IAM) user with admin permissions, as you cannot use your root account.
2. Log in into your AWS account using your newly-created IAM user.
3. Clone the repo (git clone <https://github.com/rouxelec/cdk>).
4. Run this command: source .env/bin/activate
5. Install dependencies : pip install -r requirements.txt
6. Check if CDK is installed: cdk –version

**What Does Our CDK Project Look Like?**

Once the repository is cloned, our file structure should look this:

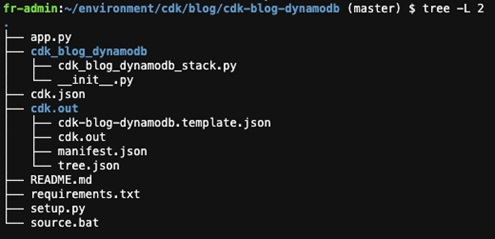
[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-3.jpg)

*Figure 3 – Project structure.*

**Amazon DynamoDB Tables**

The first project “cdk-blog-dynamodb” is to provision the Amazon DynamoDB tables we are going to use to implement the solution. It will create two tables to store information about the three VPCs and their subnets we are going to provision later on.

DynamoDB is a NoSQL database, meaning there is no specific table schema pre-defined. This is useful when you want to describe in the same table different kinds of entities like VPCs or subnets.

[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-4.jpg)

*Figure 4 – Amazon DynamoDB tables provisioning.*

Two important files for the tables creation are:

* app.py – Declares the stack you want to build.
* setup.py – Adds required dependencies.

To launch the stack, just run cdk deploy.

Note that app.py only uses one stack, so you don’t have to specify the stack to deploy.

Let’s have a look at the stack:

class CdkBlogDynamodbStack(core.Stack):

def \_\_init\_\_(self, scope: core.Construct, id: str, \*\*kwargs) -> None:

super().\_\_init\_\_(scope, id, \*\*kwargs)

# The code that defines your stack goes here

# create dynamo table

cidr\_range\_table = aws\_dynamodb.Table(

self, "cidr\_range\_table",

table\_name="cidr\_range\_table",

partition\_key=aws\_dynamodb.Attribute(

name="id",

type=aws\_dynamodb.AttributeType.STRING

)

)

last\_cidr\_range\_table = aws\_dynamodb.Table(

self, "last\_cidr\_range\_table",

table\_name="last\_cidr\_range\_table",

partition\_key=aws\_dynamodb.Attribute(

name="id",

type=aws\_dynamodb.AttributeType.STRING

)

)

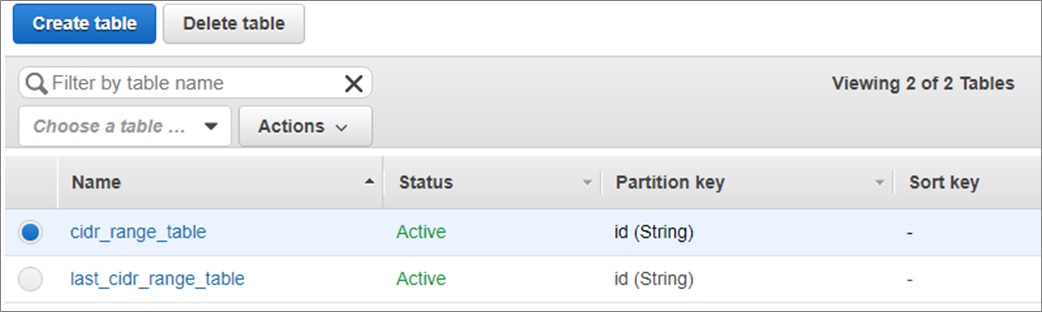
This stack is supposed to create two DynamoDB tables:

* cidr\_range\_table – Used to keep tracks on used CIDR ranges for subnets and VPCs.
* last\_cidr\_range\_table – Used to store the last used CIRDR range.

Let’s deploy the stack by running the following command:

cdk deploy cdk-blog-dynamodb --require-approval never

Once the stack status is complete, you should be able to see it through the console.

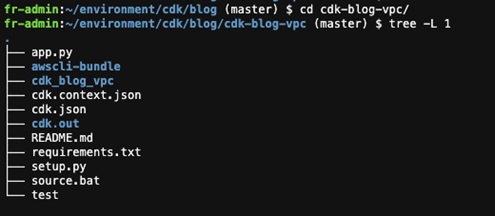
[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-5.jpg)

*Figure 5 – Amazon DynamoDB tables in the AWS console.*

**Virtual Private Clouds**

The goal here is to provision VPCs without worrying about CIDR ranges.

Let’s have a look at the stack:

[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-6.jpg)

*Figure 6 – VPC project structure.*

Now, let’s dig deeper in app.py where the VPC stacks are created.

**Stack Creation**

We can see three stacks:

* CdkBlogVpcStack – VPC stack used to provision VPCs.
* CdkBlogVpcPeeringStack – VPC peering stack used to peer two VPCs together.
* CdkBlogMyCustomResourceStack – Custom resource stack used to know when the stack is complete.
* EC2InstanceStack – Provision Amazon EC2 instance.

If we take a closer look the Custom Resource stack, we can see that we are provisioning a Lambda function:

class CdkBlogMyCustomResource(core.Construct):

def \_\_init\_\_(self, scope: core.Construct, id: str, \*\*kwargs) -> None:

super().\_\_init\_\_(scope, id)

The code associated with this Lambda function is available in the AWS Lambda folder. In this example, I might have given too much permission to simplify the overall solution, but you may want to restrict a little bit the resources for each different possible action.

The following IAM role shows the permissions used to execute the Lambda function:

my\_lambda\_role = iam.Role(self, "Role\_lambda",assumed\_by=iam.ServicePrincipal("lambda.amazonaws.com"))

my\_lambda\_role.add\_to\_policy(iam.PolicyStatement(

effect=iam.Effect.ALLOW,

resources=["\*"],

actions=["logs:\*","ec2:DescribeVpcs","ec2:DescribeInstances","ec2:DescribeInstanceAttribute","dynamodb:PutItem","ec2:DescribeSubnets","ec2:DescribeVpcPeeringConnections","ec2:DescribeRouteTables","ec2:CreateRoute","ec2:ReplaceRouteTableAssociation","ec2:CreateRouteTable","ec2:DisassociateRouteTable","ec2:AssociateRouteTable","ec2:DeleteRoute","ec2:ReplaceRoute","ec2:DeleteRouteTable"]

))

This Lambda function needs permissions to describe the newly created VPCs and their peering connections.

Note that in this example, the Lambda function created the required routes in the route tables associated of each public subnet when a VPC peering connection is active. Even if the CDK can create routes, this option has not been applied as it would create a circular dependency (the VPC peering stack has a dependency on the VPC stacks).

Adding the routes in the route tables would have made the VPC stacks creation conditional to the VPC peering stack creation, which results on a circular dependency.

**Creating the VPCs**

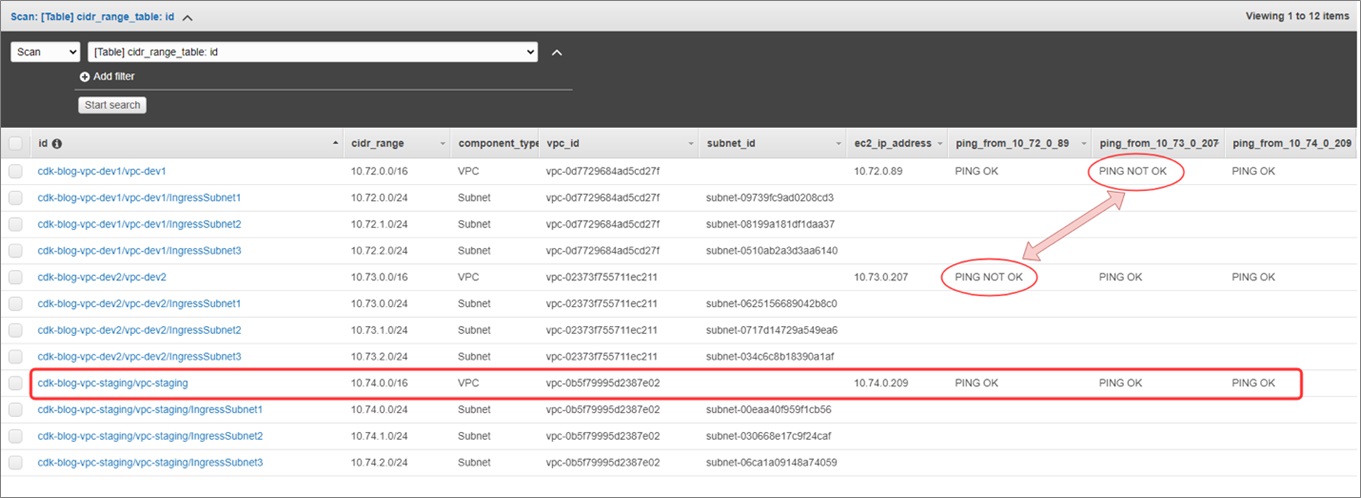
In the cdk-blog-vpc folder, enter the following command:

cdk deploy cdk-blog\* --require-approval never

Once the provisioning is complete, you should see this in the output:

cdk-blog-custom-resource2.ResponseMessage = You said "CustomResource is done"

After about one minute, the DynamoDB table should show the ping results between EC2 instances.

[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-7.jpg)

*Figure 7 – Ping between Amazon EC2 instances.*

We can see the EC2 instance deployed in the vpc-staging can ping the two instances deployed in vpc-dev-1 and vpc-dev-2. However, the instance deployed in vpc-dev-1 cannot ping the EC2 instance deployed in vpc-dev-2, and vice-versa, which proves that the non-transitivity with VPC peering.

**About Non-Transitivity with VPC Peerings**

To be able to prove the non-transitivity with VPC peering, I had to add specific routes to the route tables. Since it’s optional (demonstration purpose), I added comments surrounding the code that should be removed.

The following snippet creates the specific route:

################ WARNING #########################

# creating specific route to show no transitivity

# not required

filters = [{"Name":"tag-value","Values":["\*dev\*"]}]

vpcs = list(ec2.vpcs.filter(Filters=filters))

for vpc in vpcs:

for vpc\_id,vpc\_and\_ec2ip in vpcs\_and\_ec2ips.items():

ec2\_ip=vpc\_and\_ec2ip[1]

if not vpc.id==vpc\_and\_ec2ip[0]:

vpc\_peering\_id=vpc\_peering\_connections[vpc.id]

for route\_table in vpc.route\_tables.all():

for asso\_att in route\_table.associations\_attribute:

if not asso\_att.get('Main'):

print(ec2\_ip+"/32")

print(vpc\_peering\_id)

route\_table.create\_route(

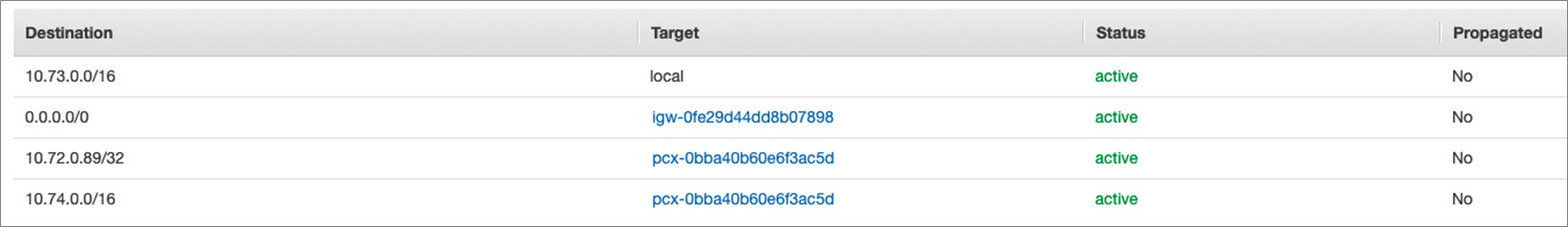
DestinationCidrBlock=ec2\_ip+"/32",

VpcPeeringConnectionId=vpc\_peering\_id

)

###################################################

In the image below, you can see that a specific route (with a /32) has been added by the code above.

[](https://d2908q01vomqb2.cloudfront.net/77de68daecd823babbb58edb1c8e14d7106e83bb/2020/09/24/Slalom-CDK-8.jpg)

*Figure 8 – VPC route table.*

**Conclusion**

In this post, you saw how easy it can be to deploy virtual private clouds using the AWS Cloud Development Kit (CDK). This solution proposes an elegant way to avoid overlapping CIDR ranges that prevent you from peering VPCs together.

This post was also a good opportunity to introduce AWS Cloud9, a powerful integrated development environment (IDE) that comes with the CDK and the AWS CLI already installed.

Moreover, this post could be used as a workshop to demonstrate the non-transitivity of VPC peering.

***The content and opinions in this blog are those of the third party author and AWS is not responsible for the content or accuracy of this post.***