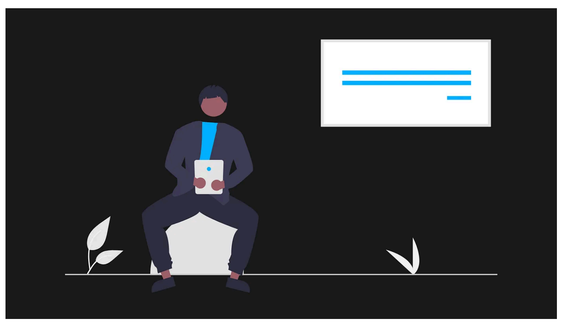
Building a Robust Serverless Messaging Service with Amazon EventBridge Pipes and CDK

Matt Carey · Follow
Published in Serverless Transformation · 10 min read · Jan 20, 2023



EventBridge Pipes is a powerful new tool from Amazon Web Services (AWS) that makes it easy to move data between sources and targets while filtering, enriching and even transforming the data en route. EventBridge used to be a singular product, the EventBridge Bus. However, in the last few months, AWS has expanded it into a complete product suite for building event-driven applications, including; EventBuses, Pipes and Schedulers.

High-scale messaging systems are traditionally complex to build. They must be able to handle a high volume of messages concurrently whilst ensuring that messages are not lost due to failures. In addition, they often require complex features such as routing, filtering, and transformation of messages, which can be challenging to implement and maintain. Pipes solves these problems by providing industry-leading horizontal scaling, redundancy with dead letter queues and inbuilt transformations, filtering and enrichment capabilities.

Using the Cloud Development Kit (CDK), we can build and deploy our messaging service without leaving our chosen development environment (in Typescript, too! Bye-bye AWS console and bye-bye writing YAML).

Example Application

We are building a web app with a fully serverless backend. We need a microservice that sends an email to the user whenever the user changes something security-related on their account, such as an address. This account data is stored in a DynamoDB table. The messaging service should construct and send an email to the user with identifying information about them and the change that has occurred.

```
ToAddresses: [userEmail]

Subject: `Notice: ${modifiedAttribute} Change Successful`

Body: `Dear ${firstName} ${lastName},

This is confirmation that your ${modifiedAttribute} for your account associated
If this is a change you made, we're good to go!

If you did not personally request this change, please reset your password or con
```

The catch is that the DynamoDB table the user has modified does not store names or email addresses, only a Cognito sub (user id). The Cognito User Pool is used as a single source of truth for security reasons and GDPR compliance. Therefore, we must query the rest of the information, such as name and email address, from Cognito directly.

> AWS Cognito is a user authentication and identity management service. It has its own data storage called User Pools which stores sign-up information. This is separate from the database, in this case, an AWS DynamoDB table.

The DynamoDB table may store a projection of the user data maintained with event-driven updates. However, the DynamoDB table data will only be eventually constant, not strongly consistent, and so cannot be relied upon apon for high-priority features.

What are Pipes?

AWS released Pipes on 1st December 2022. It makes life easier for developers to “create point-to-point integrations between event producers and consumers” while reducing the need for writing integration code. Essentially a Pipe automatically routes events from source to target, reducing the amount of extra code to write when building event-driven applications.

Less code means less maintenance and faster building times, making Pipes one of AWS’s most exciting features this year.

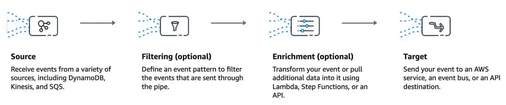
How are EventBridge Pipes different from EventBridge Buses?

Event buses are like post offices for AWS services. Different resources can send and receive messages or “events” to each other through this central service. Think of it as a middleman using forwarding rules to direct messages to the target services. For example, a user might change something on their account, which creates an event on the event bus. If the event matches a rule, then the rule target will be invoked and passed the event. This is a modern-day implementation of the Enterprise Service Bus architectural pattern.

EventBridge Pipes, on the other hand, passes events from one AWS service directly to another. They allow you to connect an event source to a target service with no extra glue code needed. In addition, you can connect a Lambda function or other AWS service to manipulate or “enrich” the events before they reach the target. Pipes is essentially ETL as a managed AWS product. Optional filtering and event transformation are available directly in the Pipe, giving almost limitless flexibility to your serverless event-driven architecture.

Why use Pipes?

The flexibility of Pipes makes it perfect for our messaging service. We have an event source needing to send an event to another AWS service. The event needs extra information fetching and adding somewhere along the flow. With Pipes, we can plug together each piece of infrastructure with minimal extra code. What’s more, these individual pieces of infrastructure do not have to be Pipe specific. As we will see later, a significant benefit of Pipes is that it allows us to reuse existing resources to create new event-driven patterns.



EventBridge Pipes as described in AWS console

Building the Architecture

To build the Pipe, we must first choose a source producing the event. In our case, this is the DynamoDB stream of the table.

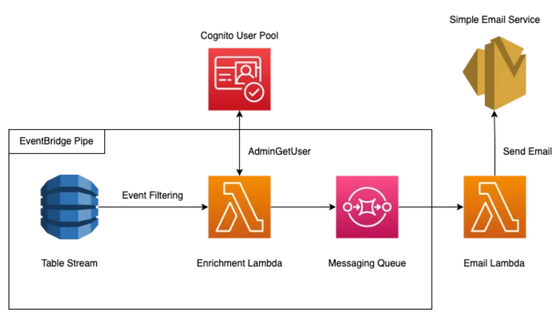
> DynamoDB Streams is a feature of Amazon DynamoDB that allows you to capture changes to the data in a DynamoDB table in near real-time.

Next, we can specify a filter to only process events in the Pipe that match the rule. AWS does not charge for events that are filtered out before the enrichment step, so it's better to do as much of the filtering here rather than later on. That being said, Pipes is super cheap. 5 million requests/month post filtering will cost you a grand total of $2.00. That is 60% cheaper than EventBridge Bus events costing a whopping $5.00 for the same number of events. Prices are correct in us-east-1 as of time of writing, see here for the current pricing per region.

To send an email, we need to get the user details, such as; first name, last name, and email, which are stored in Cognito. With Pipes, we can enrich the events using AWS Lambda, amongst several other services. In this case, we should use a Lambda function to query the user id in the Cognito User Pool. The beauty of EventBridge Pipes is that, since this is a reasonably common access pattern, we probably already have this exact Lambda function in our architecture. All that is needed is to specify the function ARN to the Pipe.

Finally, we need to pick a target destination. Pipes supports all the major AWS services, including Amazon Step Functions, Kinesis Data Streams, AWS Lambda, and third-party APIs using EventBridge API destinations as targets. In our case, we will send the event to an SQS queue. Using a queue, we can restrict message throughput and protect the email service from overloading. We can also store messages in a dead letter queue in the event of downtime of the email service.

An email Lambda which will construct and send the email with SES, can be triggered as an event source of the queue.



Architecture diagram

Infrastructure as Code (IaC)

For this project, we are using CDK for IaC. CDK allows us to write IaC in imperative programming languages such as TypeScript or Python rather than declarative ones such as YAML or JSON. This means developers can take advantage of features such as code editor support for syntax highlighting, formatters and linters. Type safety provides added security and helps catch errors easily. IaC written in these familiar languages will be more comprehensible and readable to most developers than in YAML. CDK also has the added benefit of being open source, so bugs, issues and new features are all in the public domain.

CDK released the L1 construct for Pipes in v2.55 back in December 2022. Unfortunately, the L2 construct is still in progress, so we will have to manually specify most of the CloudFormation template but bear with me, it won't be too painful. The community is working on an L2 construct; you can find the issue on GitHub here.

Please check out the example repo linked at the end of this article for the project structure. The aim with the code snippets below is to give you the information to be able to implement Pipes into your own project not a quick start guide to CDK. I can recommend this article for an intro to CDK.

DynamoDB

The table is created as shown below. Note the use of generic Partition and Sort keys "PK" and "SK" as recommended by Alex DeBrie and the inclusion of the "stream" attribute to create the DynamoDB stream. Here I have used "RemovalPolicy.DESTROY" as it is a development table. In production you would want to "RETAIN" the table.

```
const sourceTable = new Table(this, 'example-table-id', {
  tableName: 'example-table',
  partitionKey: { name: 'PK', type: AttributeType.STRING },
  sortKey: { name: 'SK', type: AttributeType.STRING },
  stream: StreamViewType.NEW_AND_OLD_IMAGES,
  billingMode: BillingMode.PAY_PER_REQUEST,
  removalPolicy: RemovalPolicy.DESTROY,
  pointInTimeRecovery: true,
});
```

Lambdas

The infrastructure for Lambda is generic so we won't include it here but please see the included repo if you are interested. The code for the enrichment Lambda handler function is found below. Here you can see the DynamoDB "NewImage" being compared to the "OldImage" to determine modified table attributes. The "getUser" function queries the "userId" in Cognito. A string is returned by the enrichment Lambda and is piped straight into the target SQS messaging queue body.

```
export const handler = async (event: DynamoDBStreamEvent): Promise<string> => {
  const record: DynamoDBRecord = event.Records[0];

  if (record.dynamodb?.NewImage == null && record.dynamodb?.OldImage == null) {
    throw new Error('No NewImage or OldImage found');
  }

  const modifiedAttributes: string[] = [];
  for (const key in record.dynamodb.NewImage) {
    if (record.dynamodb.NewImage[key].S !== record.dynamodb.OldImage?.[key].S) {
      modifiedAttributes.push(key);
    }
  }

  if (modifiedAttributes.length === 0) {
    throw new Error('No changed parameters found');
  }

  const userId = record.dynamodb.NewImage?.userId.S;

  if (userId == null) {
    throw new Error('No userId found');
  }

  const user = await getUser(userId);

  return JSON.stringify({
    ...user,
    modifiedAttributes,
  } as MessageBody);
};
```

SQS

The IaC for creating the queue is trivial.

```
const queue = new Queue(this, 'ExampleQueue', {
  queueName: buildResourceName('example-queue'),
});
```

Remember to create an event source and add the email Lambda function to it. The "batchSize" is the number of queue messages consumed by the target per target invocation. Setting it to 1 means, we only have to deal with 1 message per invocation of the email Lambda. If you are concerned about the throughput of the queue, you could increase "batchSize", but you would have to add loops to the email Lambda to process each message individually.

```
const eventSource = new SqsEventSource(props.targetQueue, {
  batchSize: 1,
});
```

```
emailLambda.addEventSource(eventSource);
```

EventBridge Pipes

Let's create a role for the Pipe to assume. We need to give permissions to read the DynamoDB stream, execute a Lambda and send a message to SQS. We will pass the sourceTable, enrichmentLambda, and targetQueue as props from other stacks.

```
const pipesRole = new Role(this, 'PipesRole', {
  roleName: 'PipesRole',
  assumedBy: new ServicePrincipal('pipes.amazonaws.com'),
  inlinePolicies: {
    PipesDynamoDBStream: new PolicyDocument({
      statements: [
        new PolicyStatement({
          actions: [
            'dynamodb:DescribeStream',
            'dynamodb:GetRecords',
            'dynamodb:GetShardIterator',
            'dynamodb:ListStreams',
          ],
          resources: [props.sourceTable.tableStreamArn],
          effect: Effect.ALLOW,
        }),
      ],
    }),
    PipesLambdaExecution: new PolicyDocument({
      statements: [
        new PolicyStatement({
          actions: ['lambda:InvokeFunction'],
          resources: [props.enrichmentLambda.functionArn],
          effect: Effect.ALLOW,
        }),
      ],
    }),
    PipesSQSSendMessage: new PolicyDocument({
      statements: [
        new PolicyStatement({
          actions: ['sqs:SendMessage', 'sqs:GetQueueAttributes'],
          resources: [props.targetQueue.queueArn],
          effect: Effect.ALLOW,
        }),
      ],
    }),
  },
});
```

Next, in the same stack, let us specify the Pipe itself. Note the "sourceParameters" attribute. Some targets also require extra "targetParameters". Here we set the batch size of the stream as 1 for convenience for testing, but as long as it's less than the batch size of the target queue, you'll never have an issue.

```
new CfnPipe(this, 'MessagingPipe', {
  name: 'MessagingPipe',
  roleArn: pipesRole.roleArn,
  source: props.sourceTable.tableStreamArn,
  sourceParameters: {
    dynamoDbStreamParameters: {
      startingPosition: 'LATEST',
      batchSize: 1,
    },
  },
  enrichment: props.enrichmentLambda.functionArn,
  target: props.targetQueue.queueArn,
});
```

How could the Architecture be Improved?

Filtering Options

Filtering of the source event in CDK is available under the "sourceParameters" props for CfnPipe. An example of how you might filter events to only leave those where one or more attributes are modified and "someAttribute" is present in the "NewImage" of the table can be seen below.

```
{
"eventName": [{
 "prefix": "MODIFY"
 }],
  "dynamodb": {
    "NewImage": {
      "someAttribute": {
        "S": [{
          "exists": true
        }]
      }
    }
  }
}
```

This will remove the vast majority of events but will still leave some filtering to be done in the enrichment Lambda function. AWS actually open-sourced the event-ruler which powers these filters. You can find the repo here.

In the future, I hope the EventBridge team and the open-source community will improve the filtering capabilities to allow you to compare exact table values. Being able to do something like this below would be really handy to avoid overpaying for events you don't want.

```
{
"eventName": [{
 "prefix": "MODIFY"
 }],
  "dynamodb": {
    "NewImage": {
      "someAttribute": {
        "S": [{
          "not-equals": <$.OldImage.someAttribute.S>
        }]
      }
    }
  }
}
```

Best Practice

Speaking with the AWS Pipes team, they told me they would prefer if users return JSON instead of strings from the enrichment Lambda and use transformations available in the console to parse the event. However, I'm not a fan of the console, especially for large applications with multiple environments, so I'm waiting for the transformation attribute to be available in CloudFormation and CDK before the swap to do this.

Future Improvements to Pipes

The next feature we are waiting for is CloudWatch logs for Pipes. We've been informed it's a priority and is actively being worked on. It should dramatically improve the developer experience working with EventBridge Pipes. The same goes for cross-account capabilities. We build a lot of multi-account architectures, so using a Pipe instead of a direct Lambda invocation would simplify them in many cases.

Summary

In conclusion, Amazon EventBridge Pipes offers a practical solution for building a robust serverless messaging service. Pipes allow developers to easily connect event sources and targets while filtering and enriching events, using existing infrastructure and reducing the need for additional code and maintenance. For example, it is possible to connect a DynamoDB stream as the event source, filter and enrich the events with information from Cognito using a Lambda function, and then send the event to an SQS queue for email delivery. **This flexible and powerful feature makes EventBridge Pipes ideal for building serverless, event-driven architectures.**

Resources and Code

- The example repo
- Open-source patterns with EventBridge Pipes
- Amazon EventBridge Pipes Documentation

Eventbridge | Eventbridge Pipes | Serverless | AWS | Microservices