CDK Day May 2022 - Track 2

CDK Day
2.15K subscribers

Subscribe

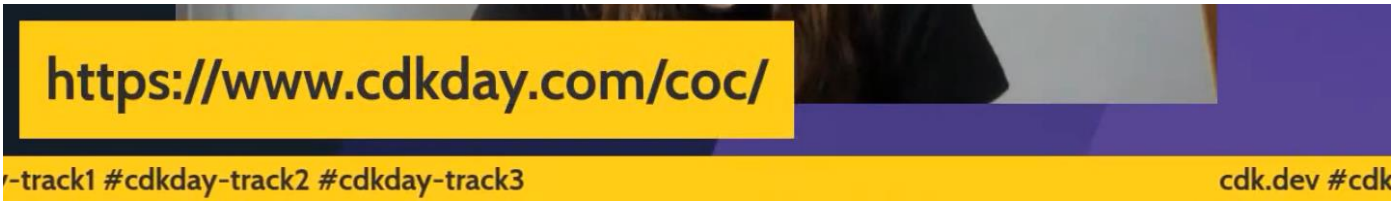
32 | Share | Download | Clip | Save

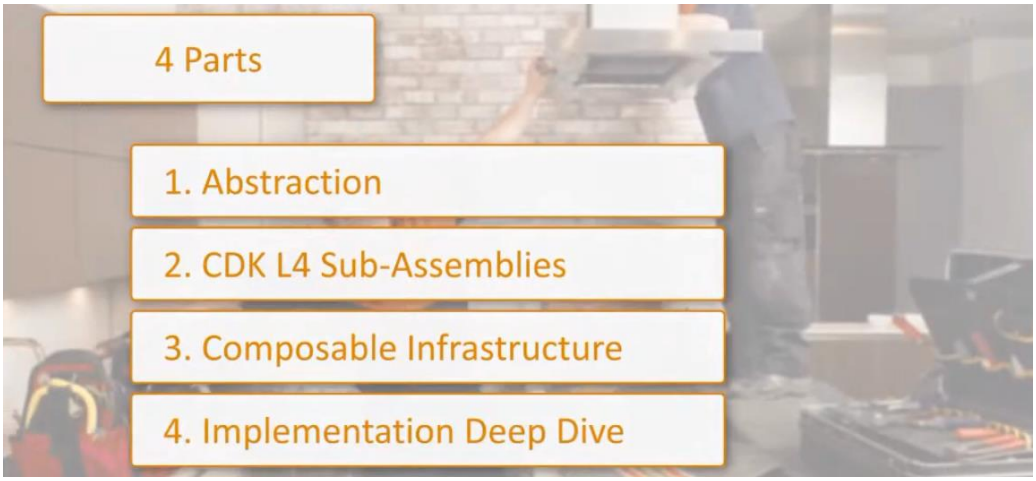
3,026 views Streamed live on May 26, 2022

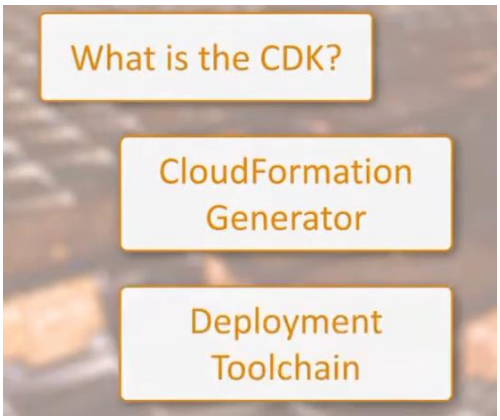
Cloud Development Kit (CDK) is a developer tool built on the open source Constructs model. CDK Day is a one-day conference that attempts to showcase the brightest and best of CDK from AWS CDK, CDK for Terraform (CDKtf), CDK for Kubernetes (cdk8s), and projen. Let's talk serverless, Kubernetes and multi-cloud all on the same day.

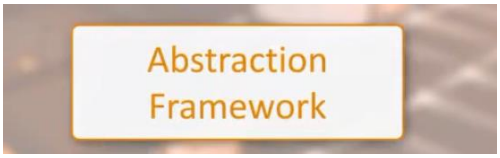
This is the livestream for Track 2 of CDK Day, featuring talks from Ansgar Mertens, Jenna Krolick, Bill Penberthy, and more. For the full agenda, please see https://www.cdkday.com/.

For the livestream of talks from Track 1, please head over to https://bit.ly/cdkday2022-track1
For the livestream of talks from Track 3, please head over to https://bit.ly/cdkday2022-track3

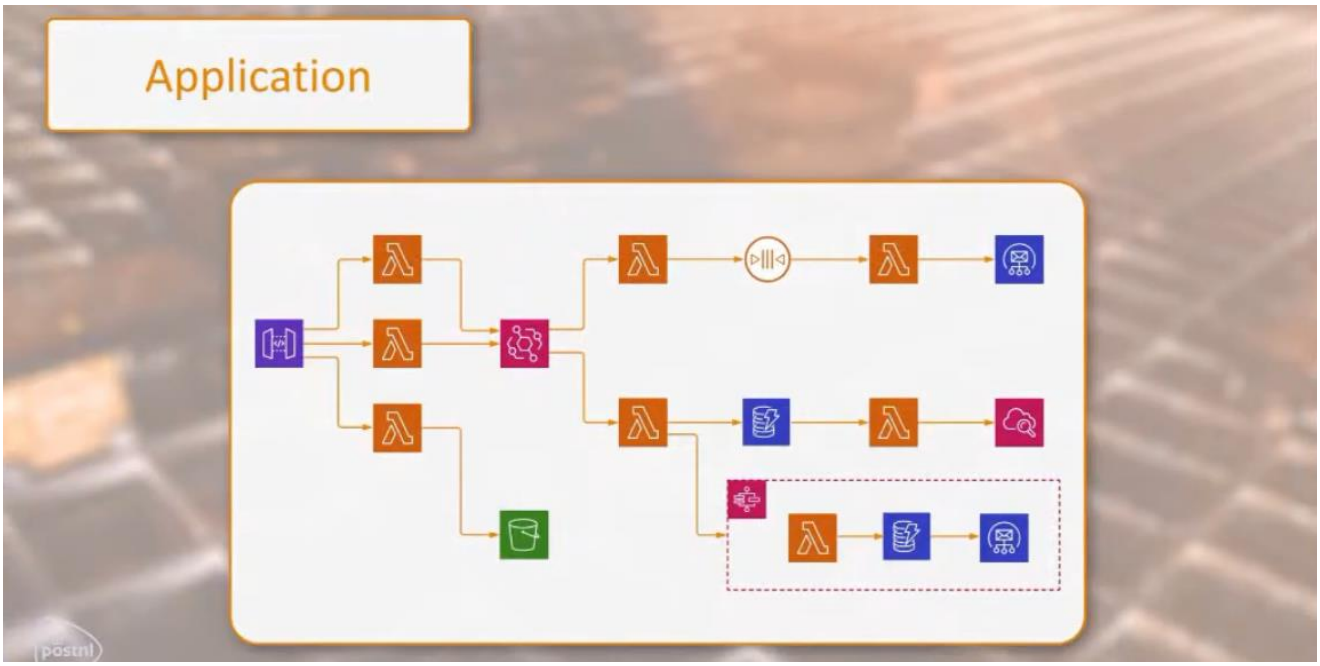




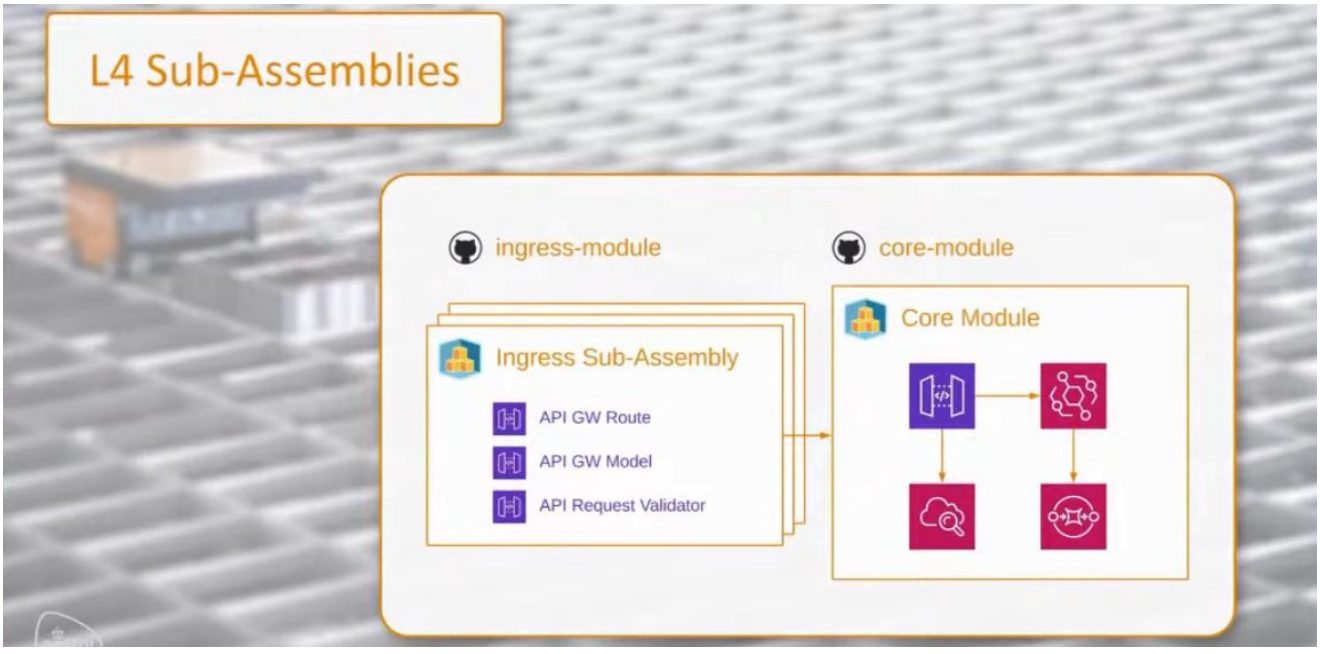




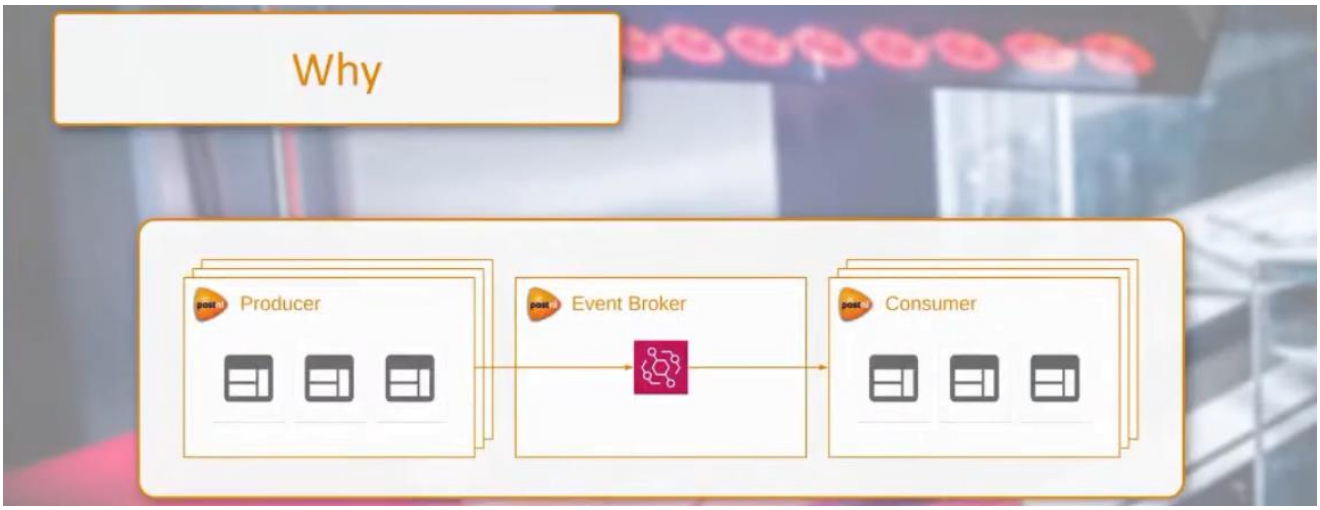
The tools we use to build these abstractions are called Constructs



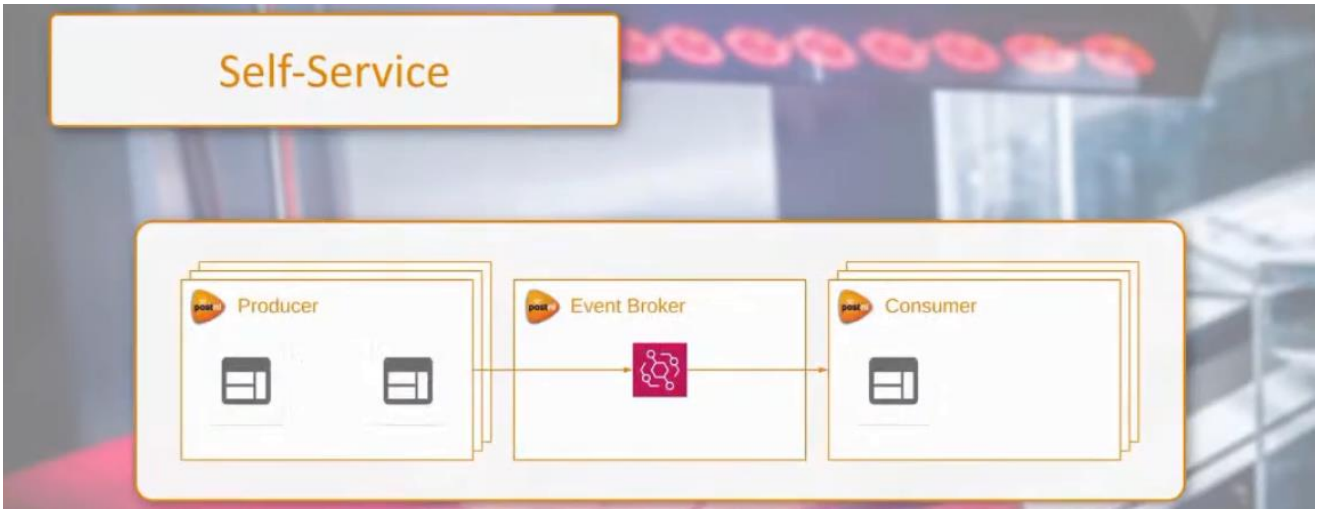
Applications are generally stored as a single CDK project in Git.

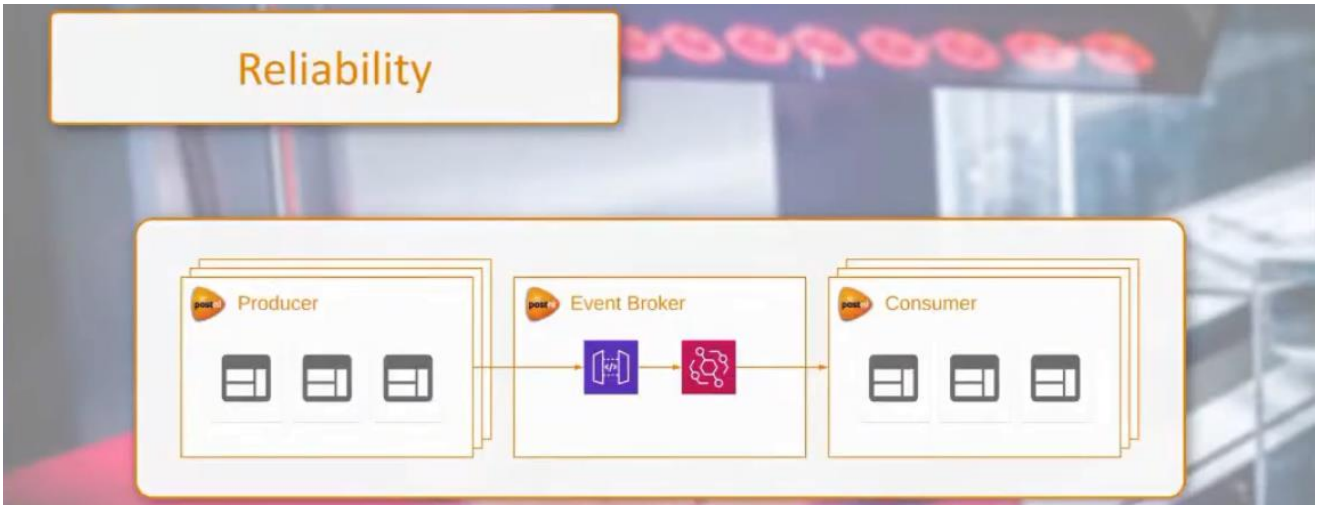


Is a standalone CDK project that can and will be deployed as a standalone stack, it will not deliver complete functionality by itself but will integrate with other components in the core module that live in their own CDK projects and CF stacks.

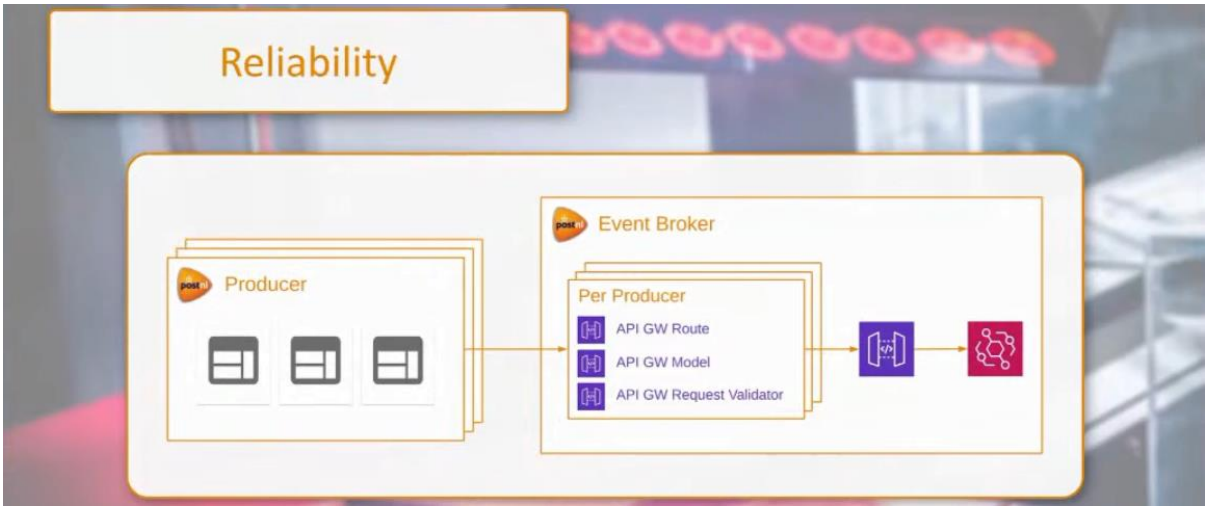


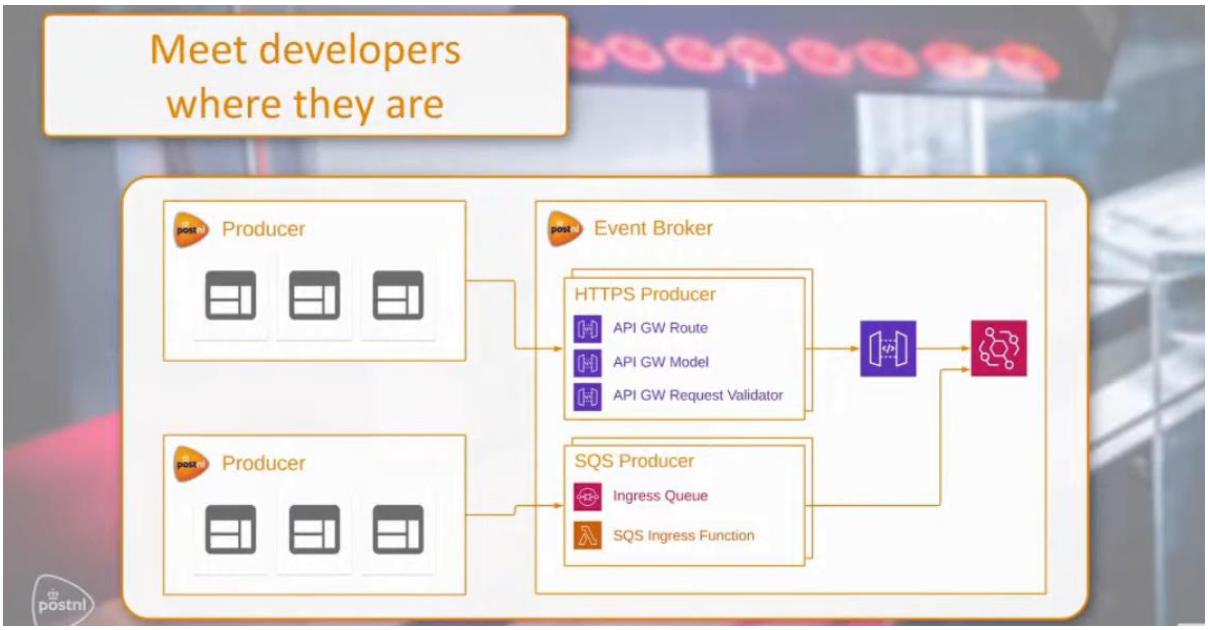
We have a central event broker that is based on the EventBridge, this allows all apps to publish events and subscribe to events in a decoupled manner.

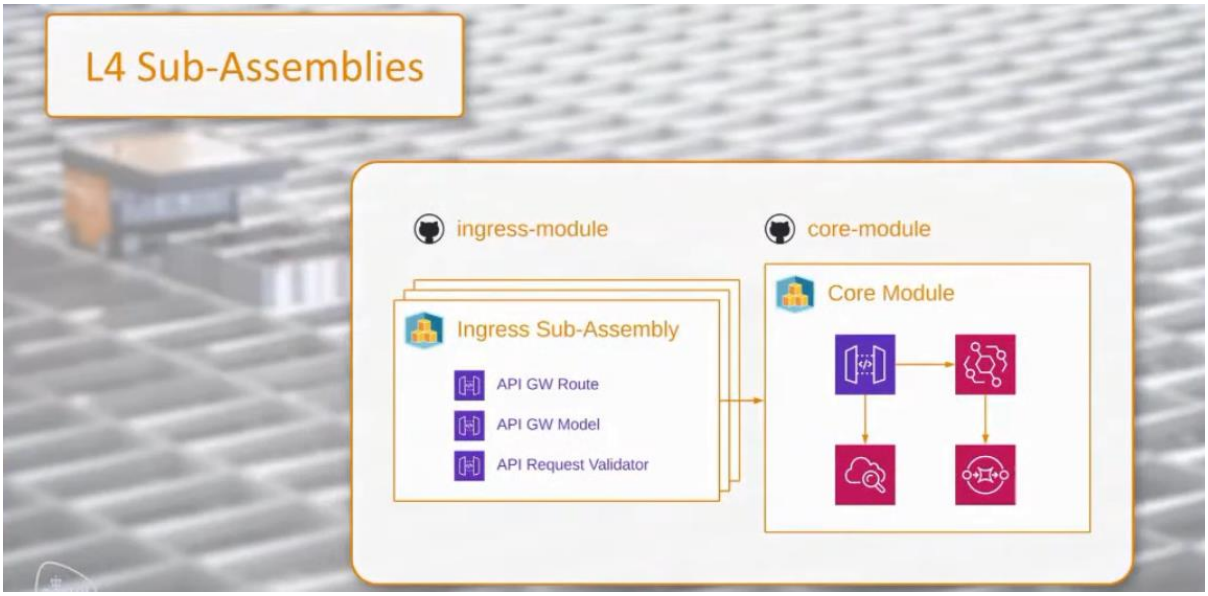




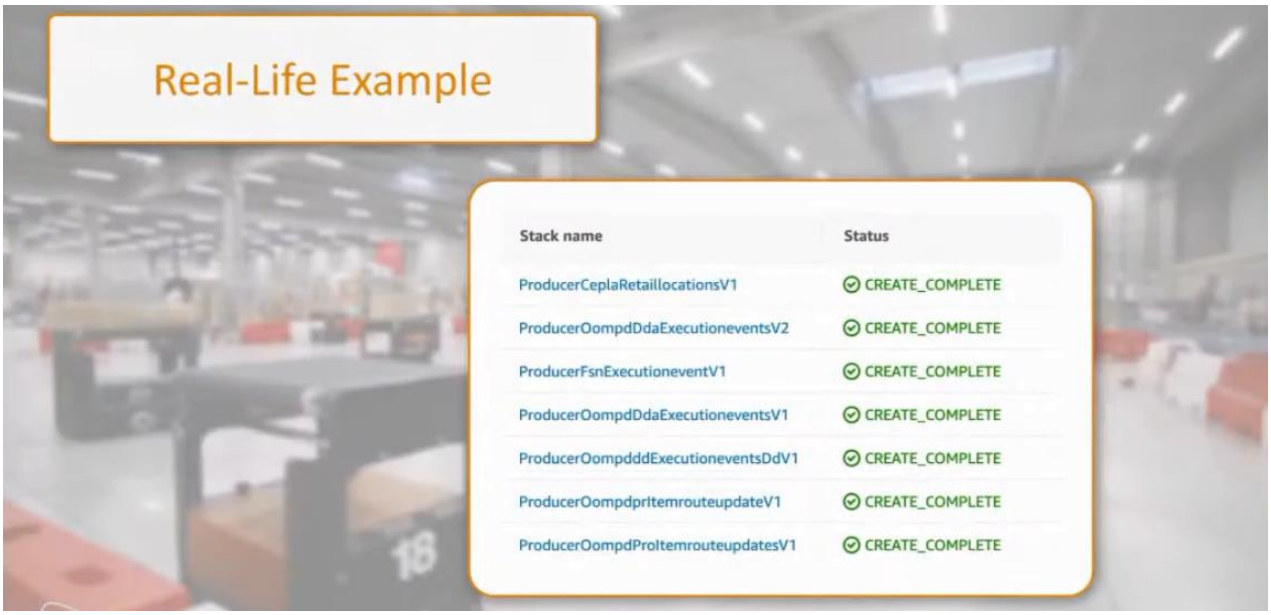
To guarantee reliability, we route events through an API Gateway that has a request validator that rejects invalid events.



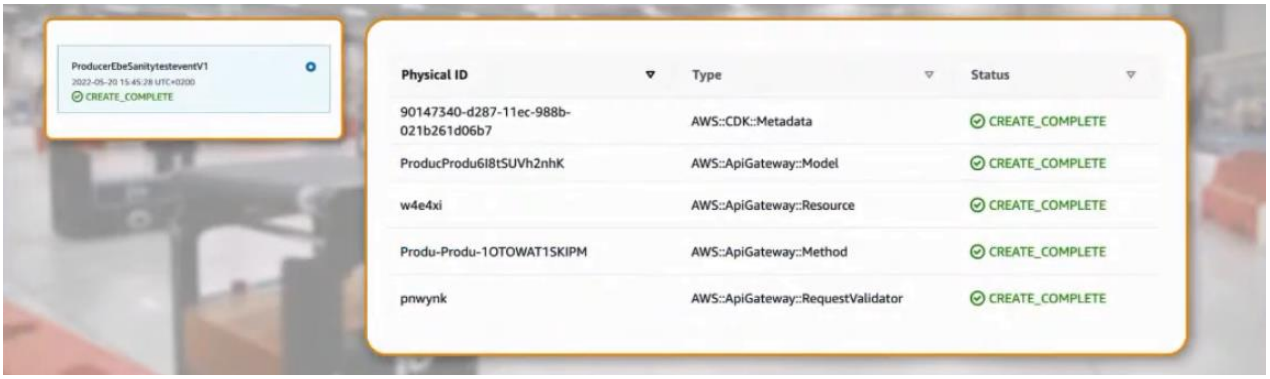




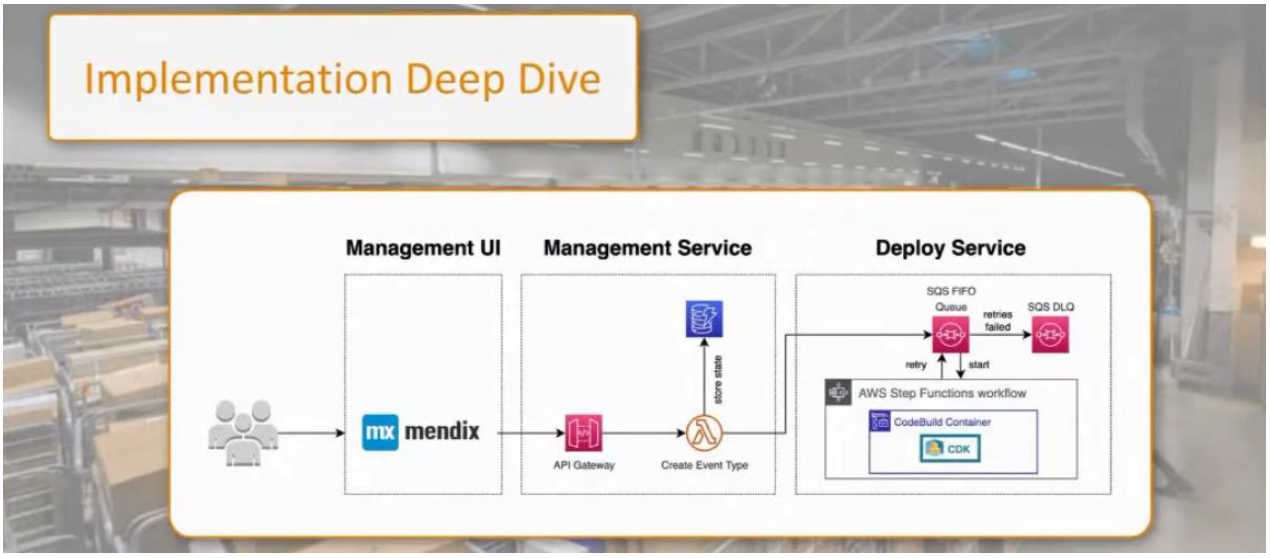
We use the L4 Sub-Assemblies to deploy the relevant components per Producer, whether they need the HTTPS Producer component or the SQS Producer component. L3 Patterns are part of a larger application or service but the L4 Sub-Assemblies are standalone stacks or a full blown CDK application that does not deliver a full functionality by itself.



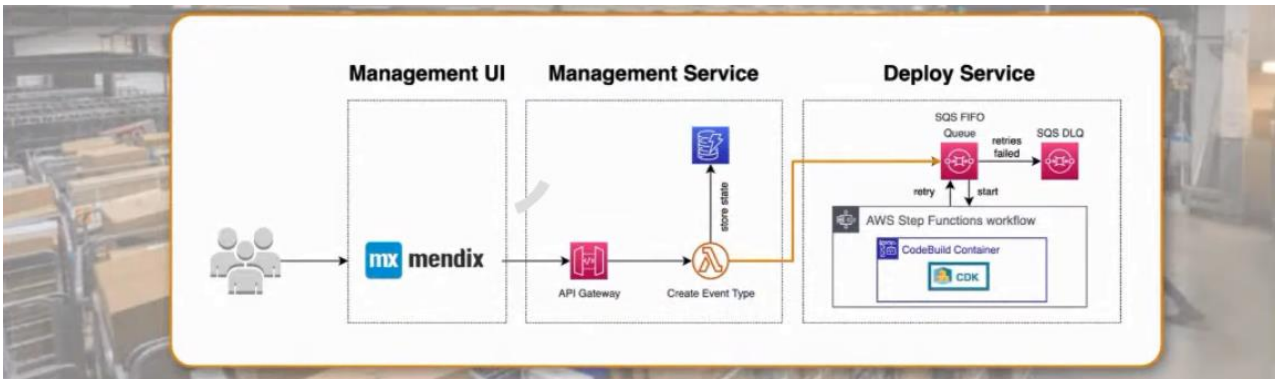
The above stacks are part of an L4 Sub-Assembly



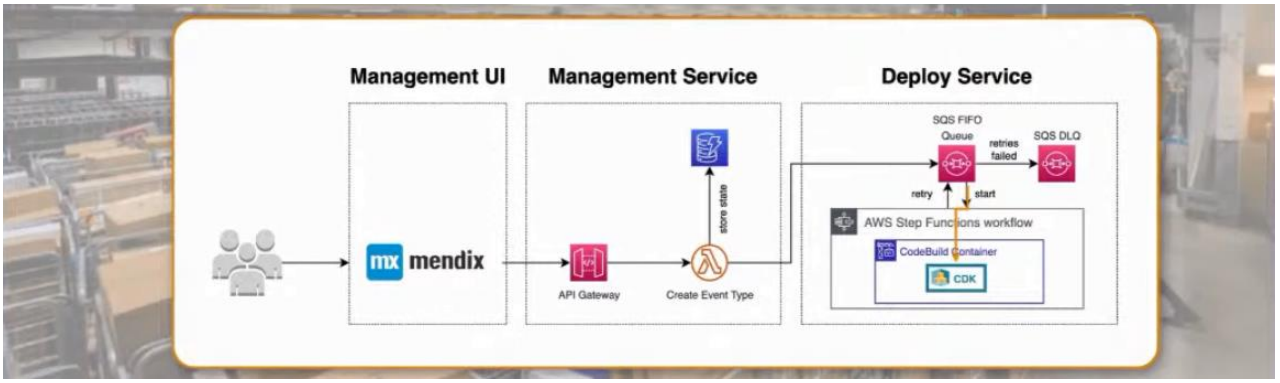
Each stack has a small number of resources needed like the ones above needed for ingesting events from the API Gateway.



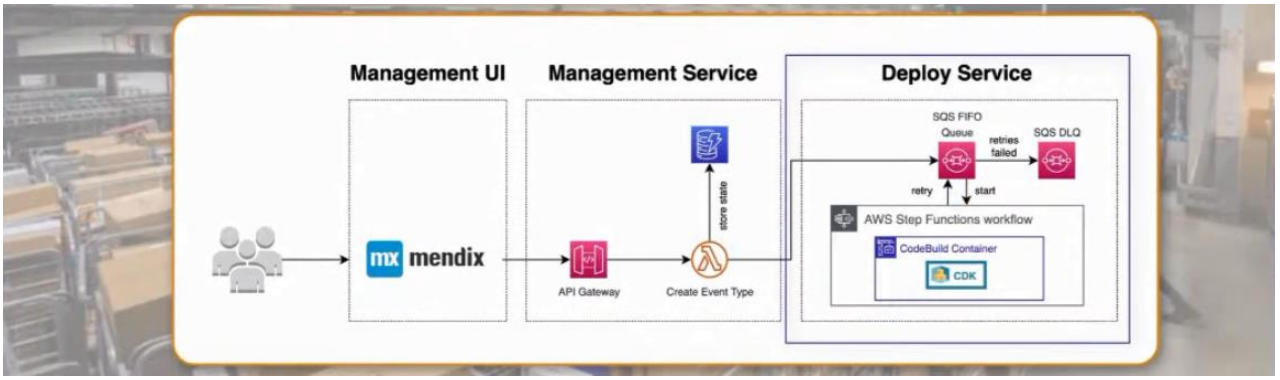
This shows an overview of the entire deployment process. It starts with a user registering an event in our management UI, the request is received by the API Gateway which authorizes the user, the request is then forwarded to a Lambda function that verifies the request itself.



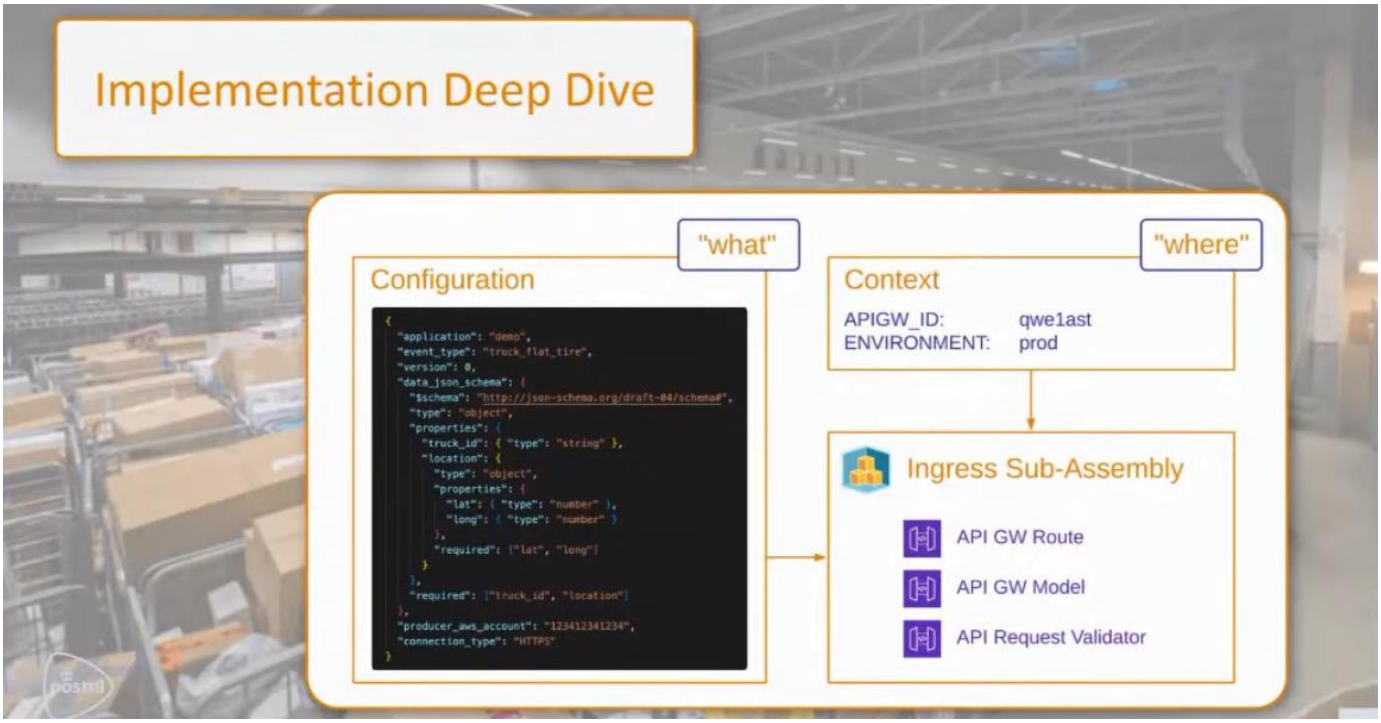
Once verified, the request is forwarded to the Deploy Service as an SQS message



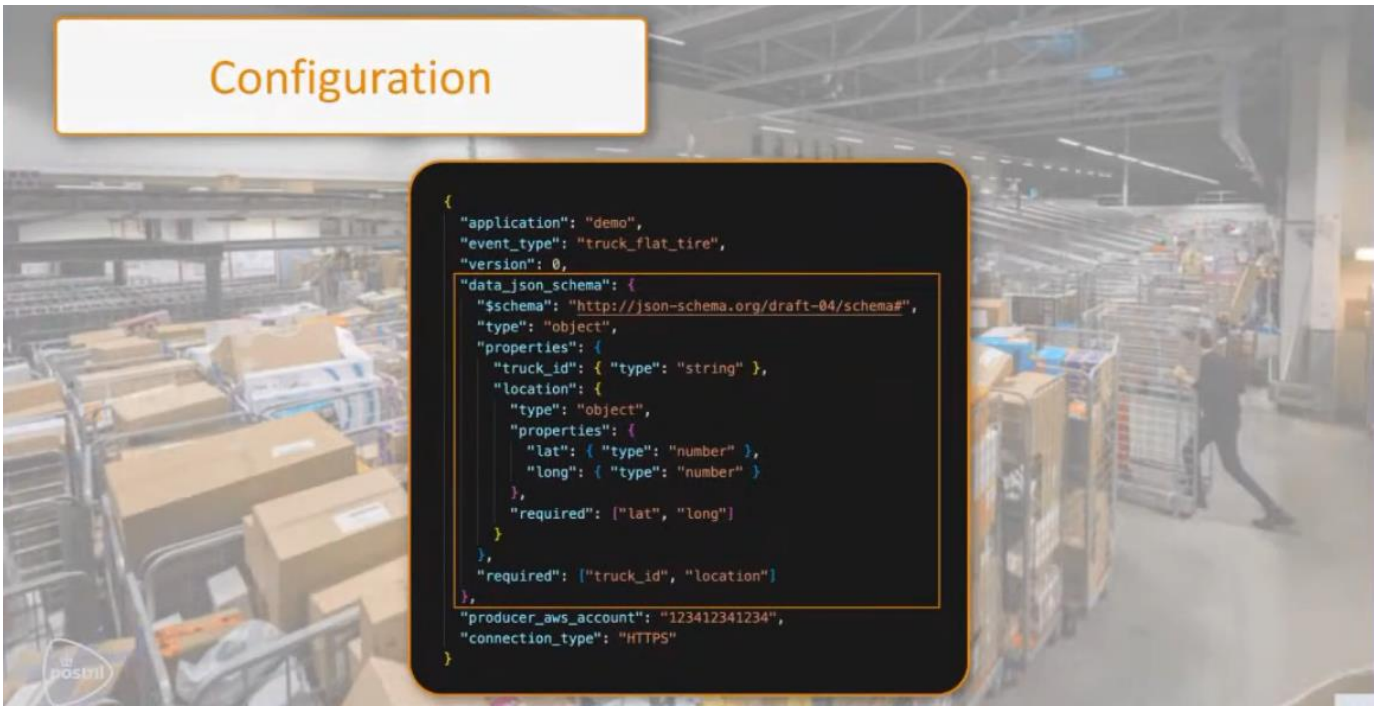
The SQS message then triggers a state machine/AWS Step Functions workflow, then through AWS CodeBuild it finally triggers the CDK.



Let us focus on the Deploy Service and start at the CDK -> Step Functions workflow -> SQS.



The CDK project is an abstraction of the resources required for a Producer. To deploy these components, the project needs to know 2 things. **What** to deploy, which mentions the specifications of the event as provided by the user. These specifications will be different for every deployment, we will call this the **sub-assembly configuration**. Next, we need to know **Where** to deploy the project. This includes things like the AWS Account ID, environment name, and the identifiers of the core components the sub-assembly should integrate with. These values will be the same for every deployment which we will call the **sub-assembly context**. The **What** part is provided to the CDK as a configuration file in JSON, the **Where** part is provided through environmental variables.



This contains all the details provided by the Producer such as the app name, the event type, the schema that defines the event and the ingress connection type.

The **Context** is a collection of environment variables which defines what the sub-assembly should interface with, above context includes the name of the API Gateway and the environment we want to deploy this sub-assembly into. These values are the same for every sub-assembly being deployed in our environments.

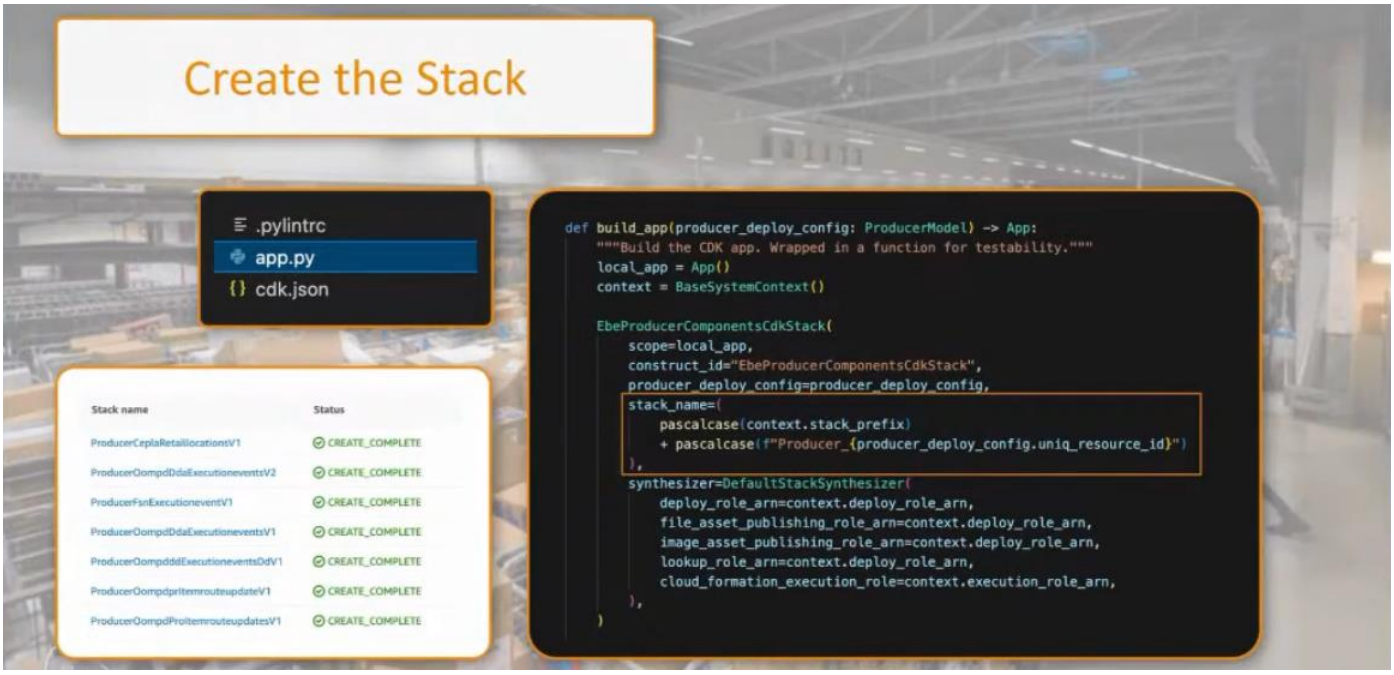
In the main section, we start by loading the configuration JSON file as a Producer model to retrieve the dictionary values from it.

The values are then loaded into the next function **cls.model_from_dict** that takes the dictionary and turns it into a Producer model that we then can reuse in the rest of the application.

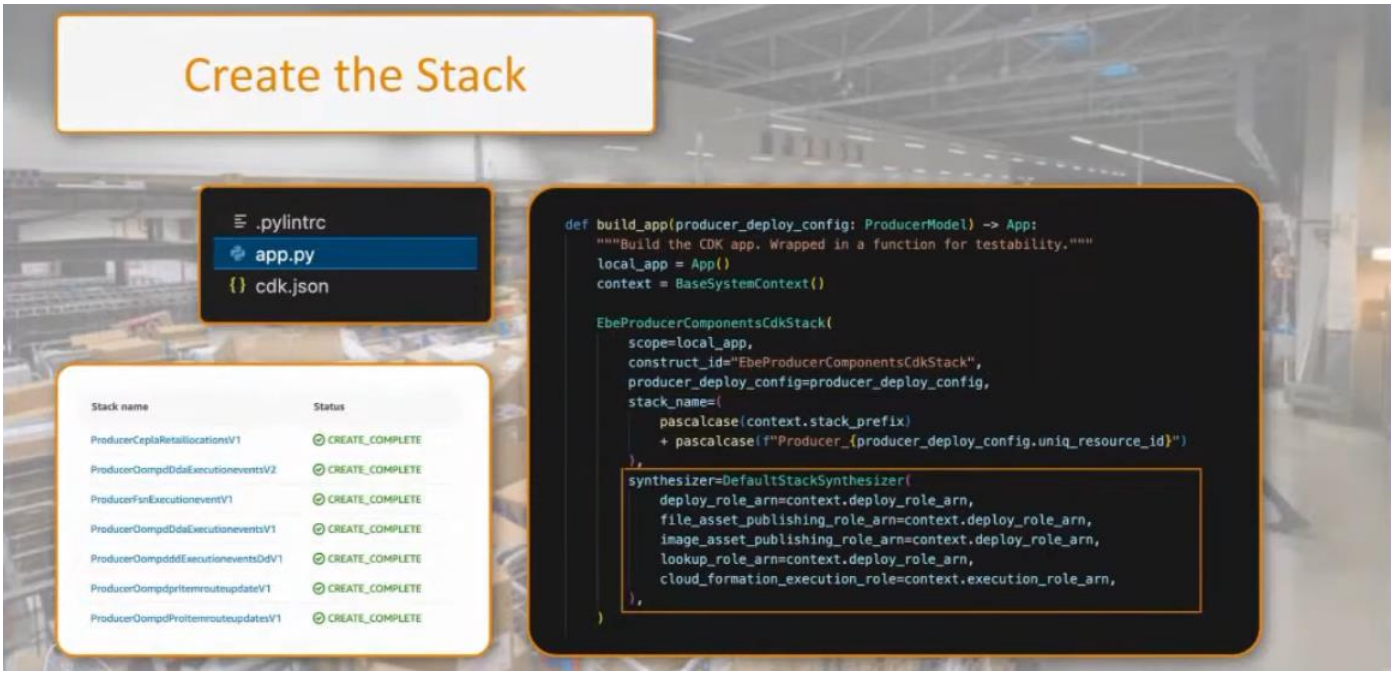
We determine the type of Producer and then return a specific model based on its connection type.

Still in the app.py, we load the environment variables and store them in a system context object/class.

Finally, we then create the stack that we want to deploy.



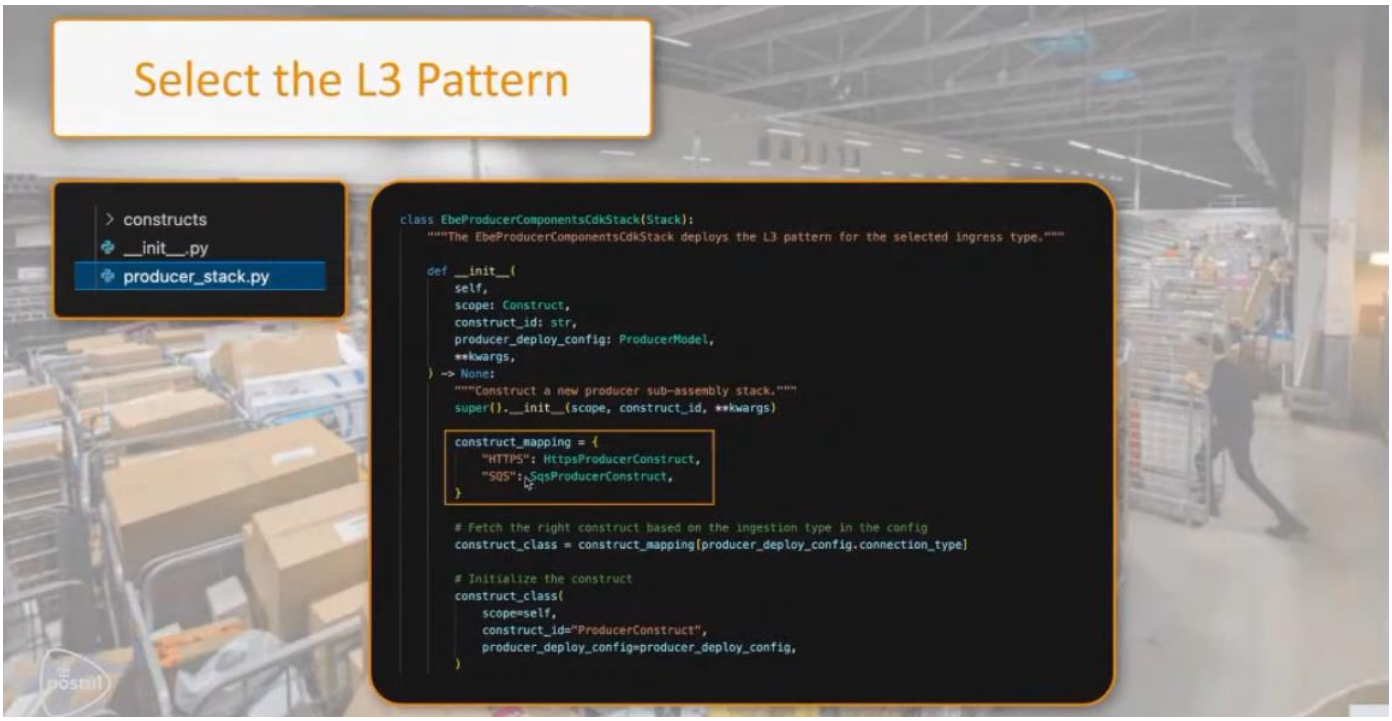
We dynamically determine the stack name that allows us to dynamically generate stack name (different for each sub-assembly). This allows us to deploy the sub-assembly many times within the same environment.



We overwrite the deploy roles used by the CDK, these roles will later be provided by AWS CodeBuild.

In this stack, we provide the Producer model as a parameter.

We looked at which L3 pattern to deploy based on the connection type in the configuration,

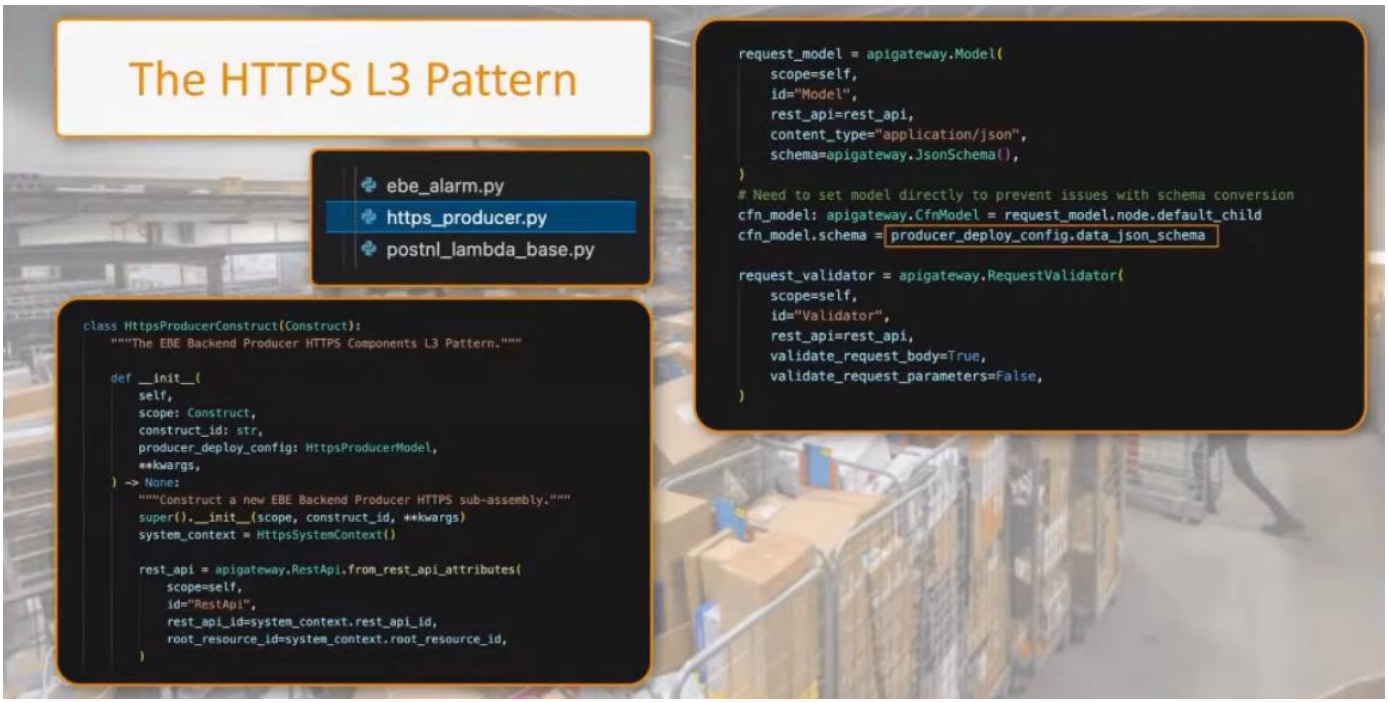


We used a mapping to map the connection type to a L3 pattern.

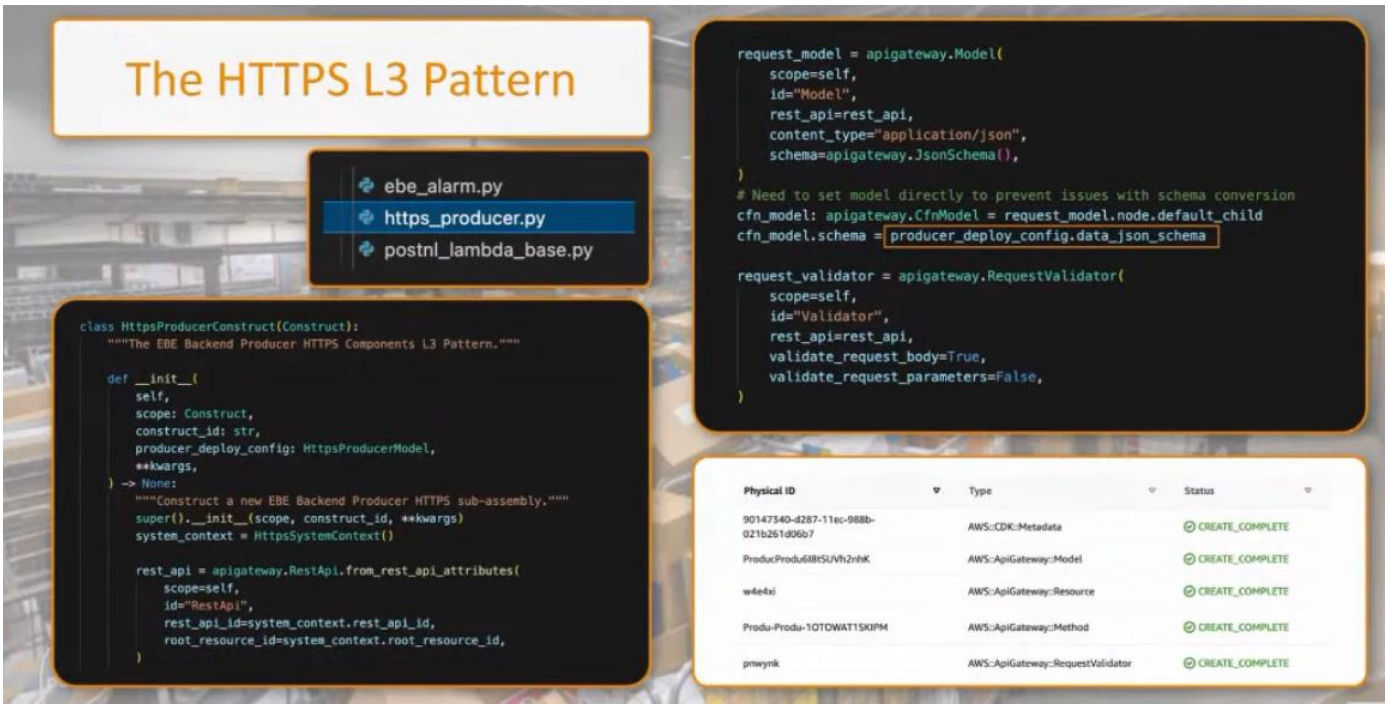
We then initialized that L3 pattern using the **producer_deploy_config** as a configuration parameter.

Finally, in the CDK, we actually deploy the L3 pattern. We log the context/environmental variables.

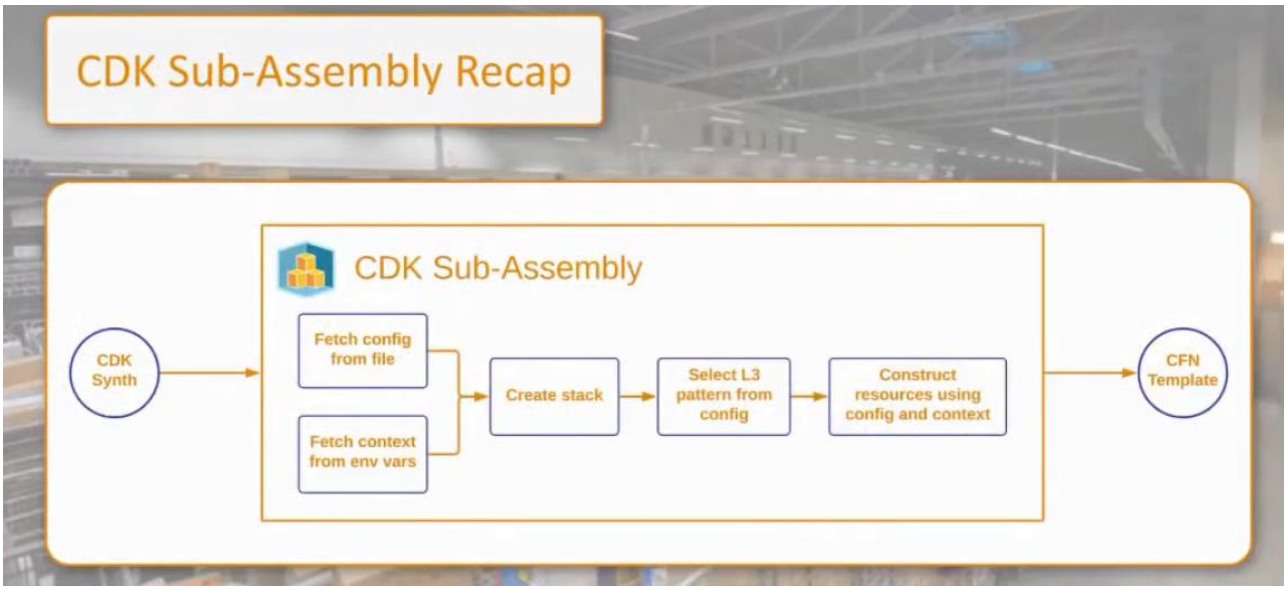
Then we import an existing API Gateway using the REST API ID and the root resource ID provided to the context.



We create a number of resources like the API Gateway model and the API Gateway request validator, wen then used the configuration supplied through the configuration file.

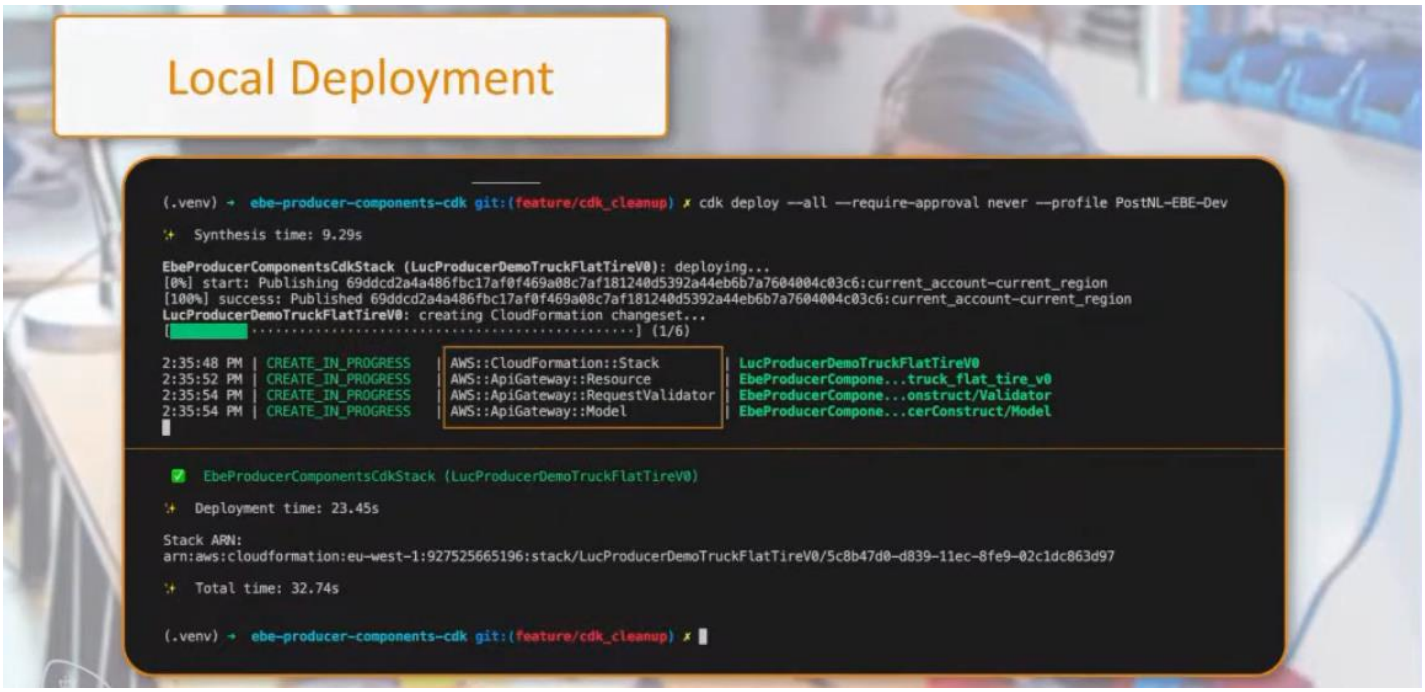


These resources translate directly to the resources being deployed by CloudFormation

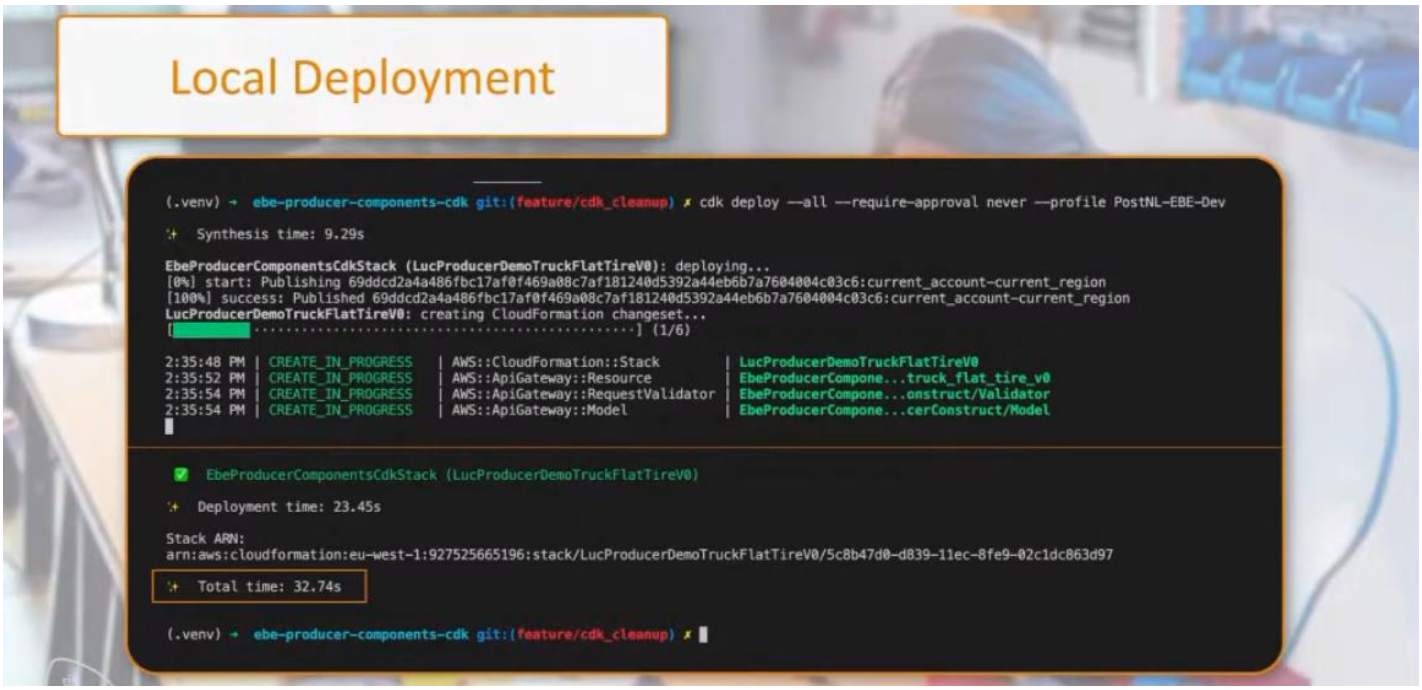


The **CDK responsibilities** includes:

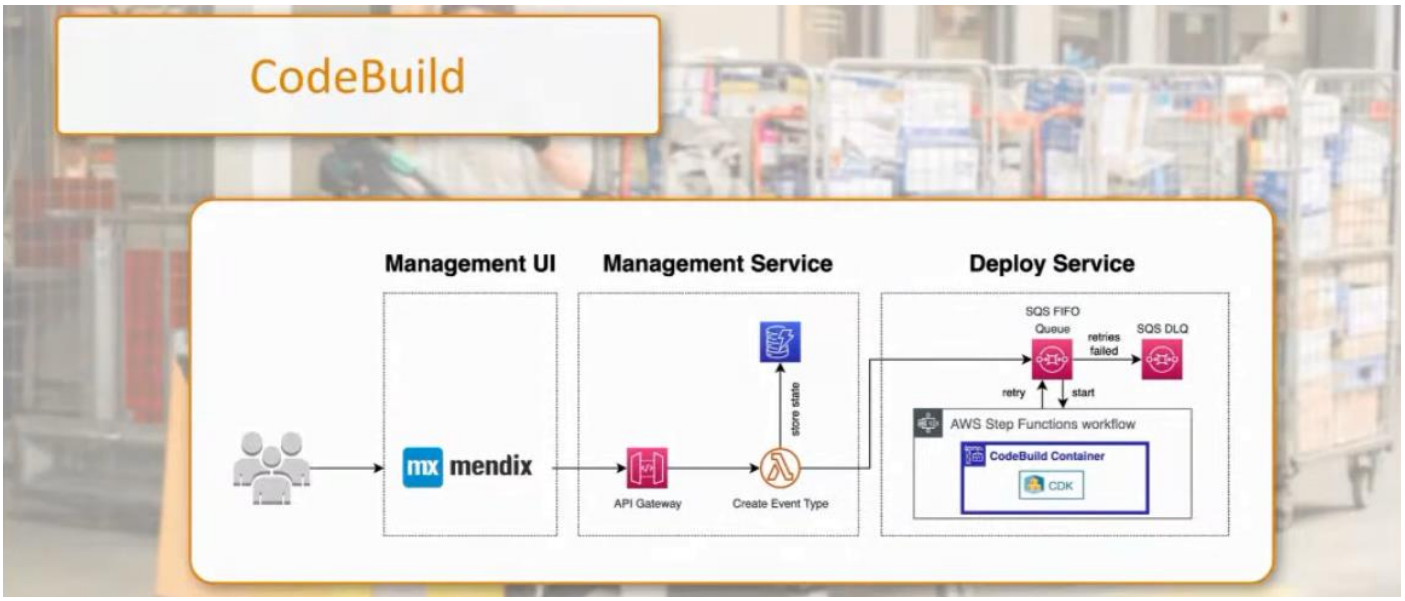
1. Fetching the config file and context and store them in objects.
2. It then creates a CF stack using values from the configuration to determine the stack name.
3. It then determines which L3 pattern to use based on the ingress type specified by the producer.
4. Finally, it constructs the resources in that pattern using the configurations specified.



To the CDK project, the configuration file is just a JSON file in a predefined location. The context environment variables are just the variables available through the OS. This allows us to do local deployments which we use during testing and validation.

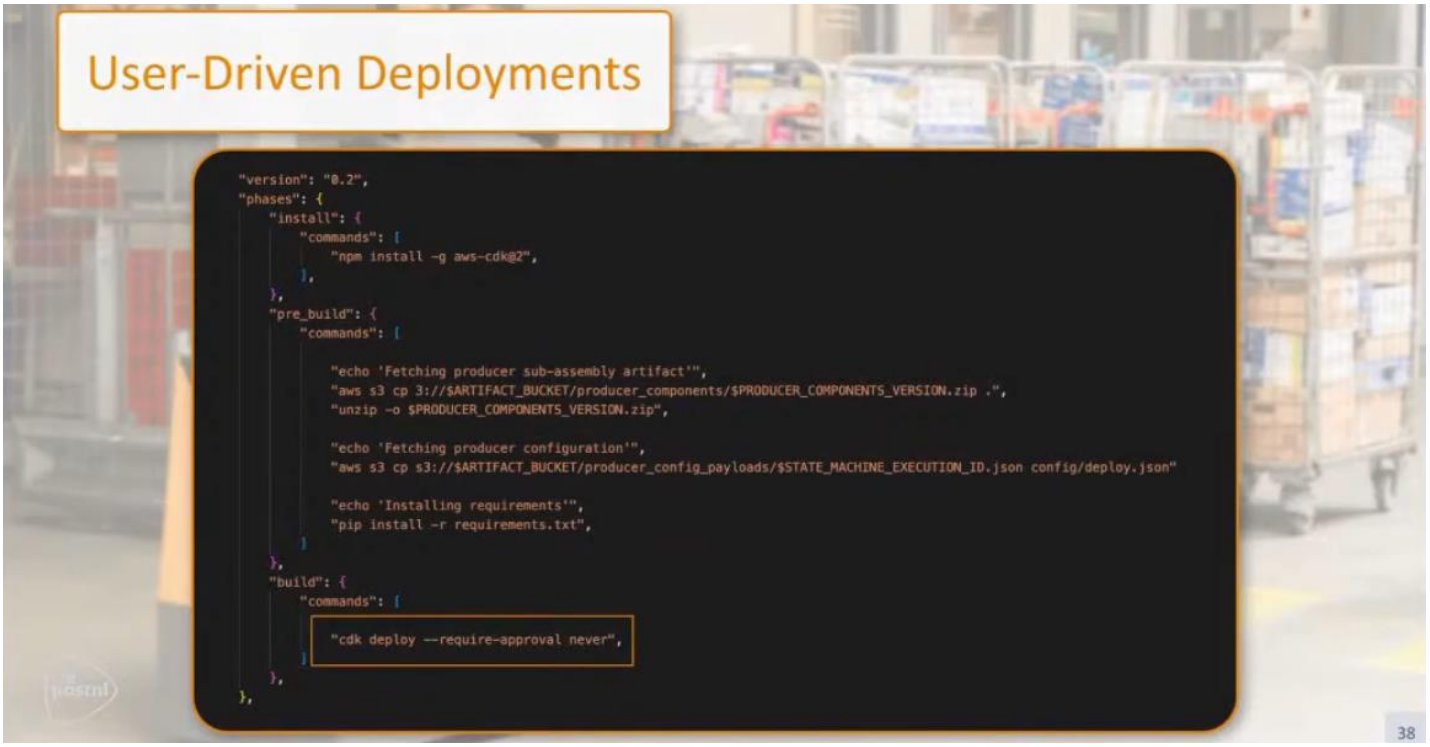


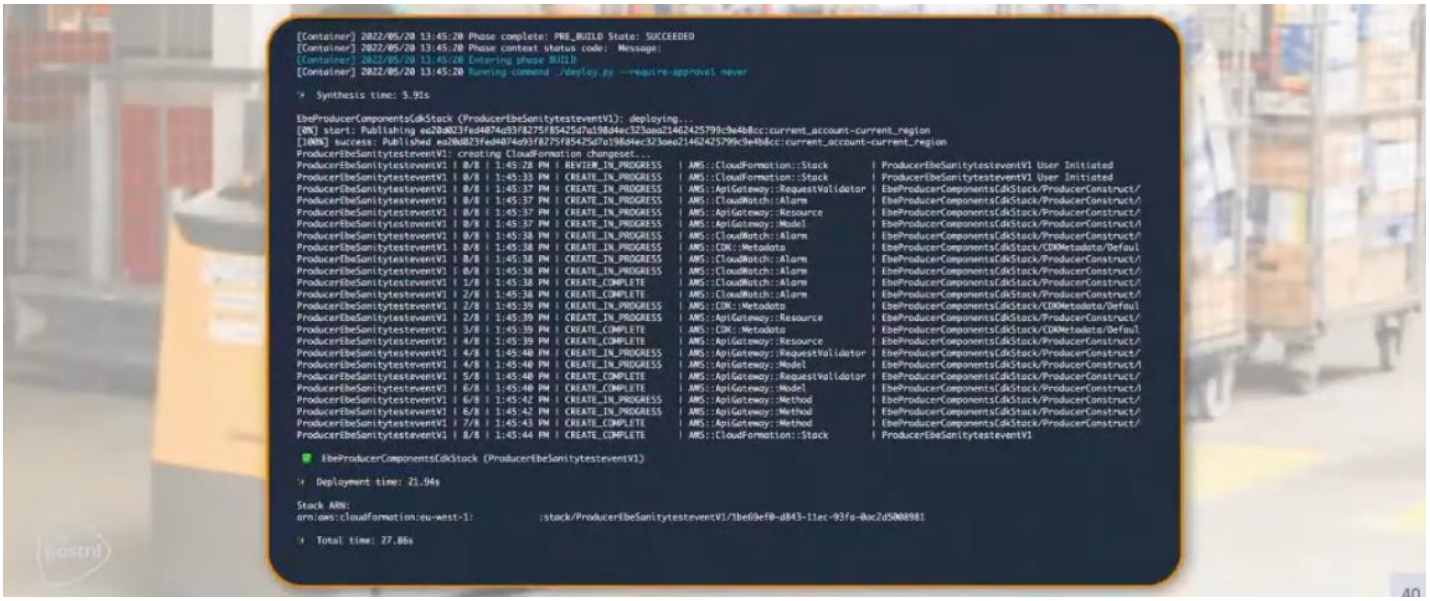
But a local deployment isn't very user-driven, we need a built environment that can be triggered by a user request. AWS CodeBuild is a perfect use-case for this because it allows us to spin up containers for a one-time job like a CDK deployment.



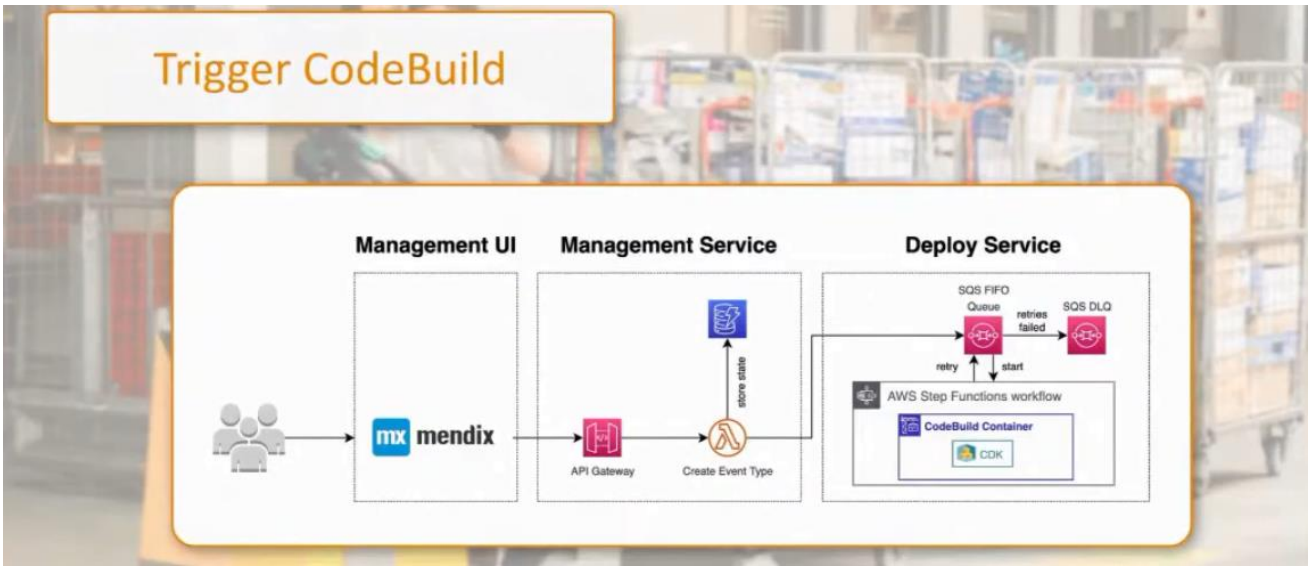
Let us look at how to invoke the CDK project on the lower right corner.

The CodeBuild container's responsibilities are defined in a buildSpec script that runs when the container is started.

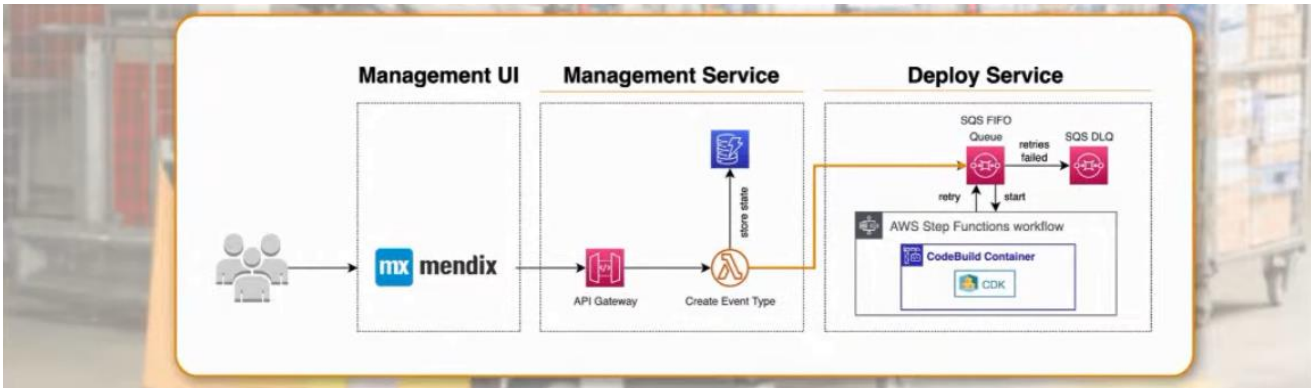




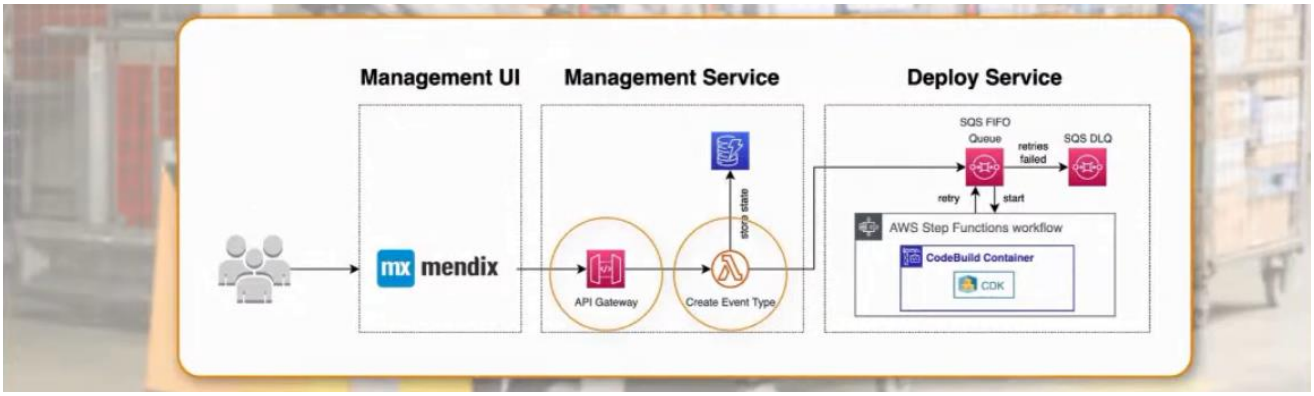
We then run the CodeBuild container to deploy a new sub-assembly.



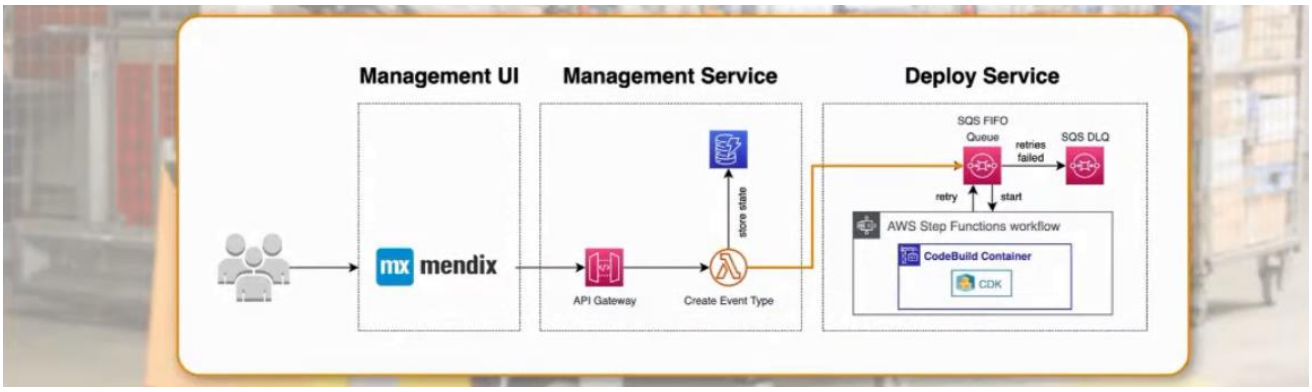
We have seen how to use CodeBuild to deploy the sub-assembly, but the user still does not know how to trigger CodeBuild. Next, let us see how a user can invoke a sub-assembly deployment.



The interface between the Management Service and the Deploy Service is an SQS message.

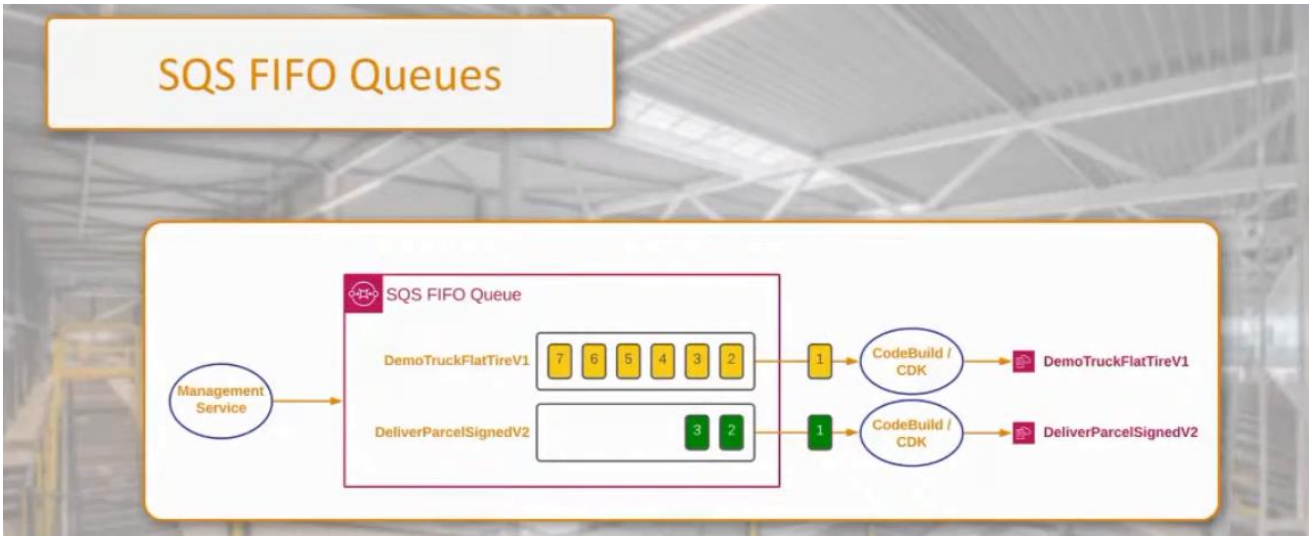


When a user makes a UI request via the Management UI, the Management Service authorizes the user and then validates their request.

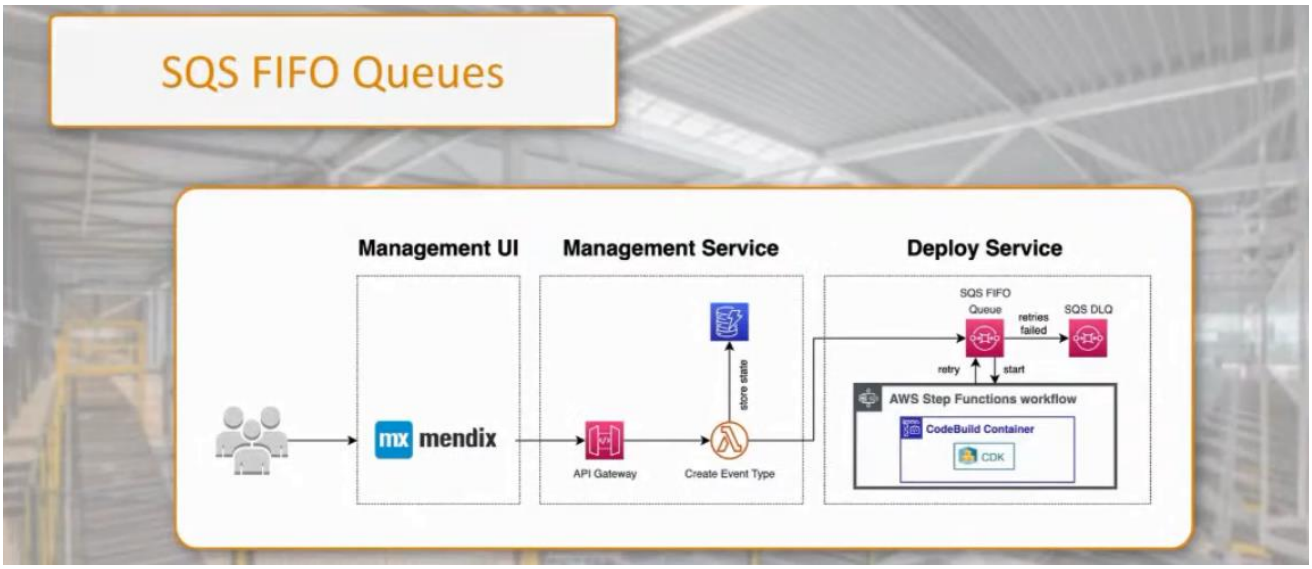


When the user and request (event) are validated, the Management Service then generates the SQS message and puts it on the FIFO queue.

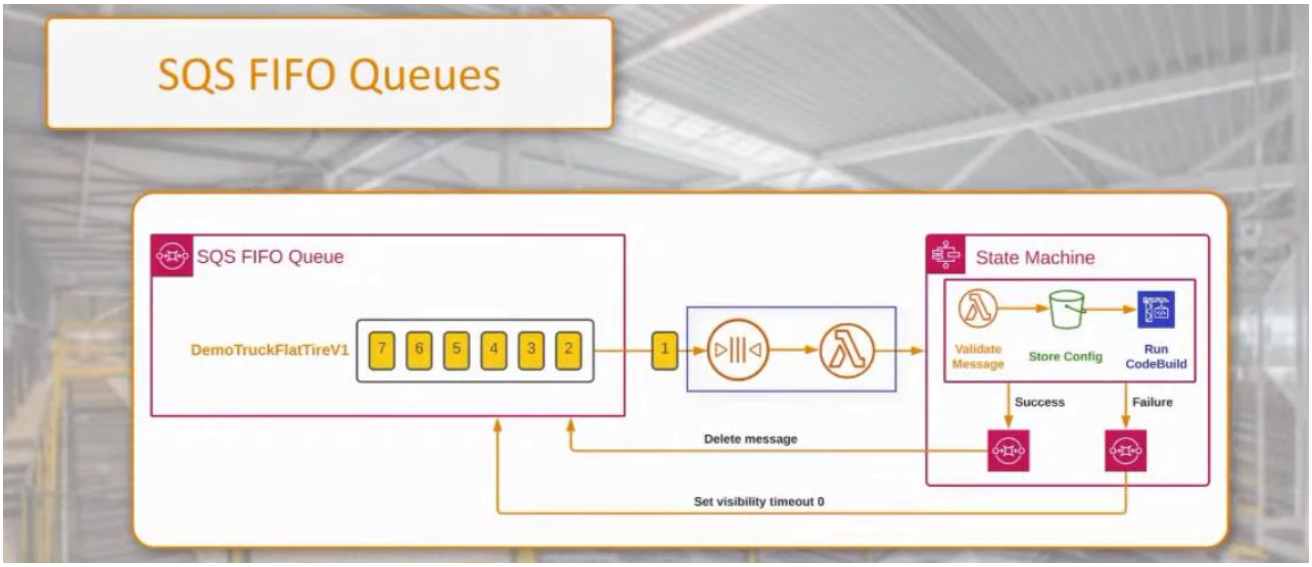
Let us see how SQS and Step Functions are used to trigger CodeBuild.



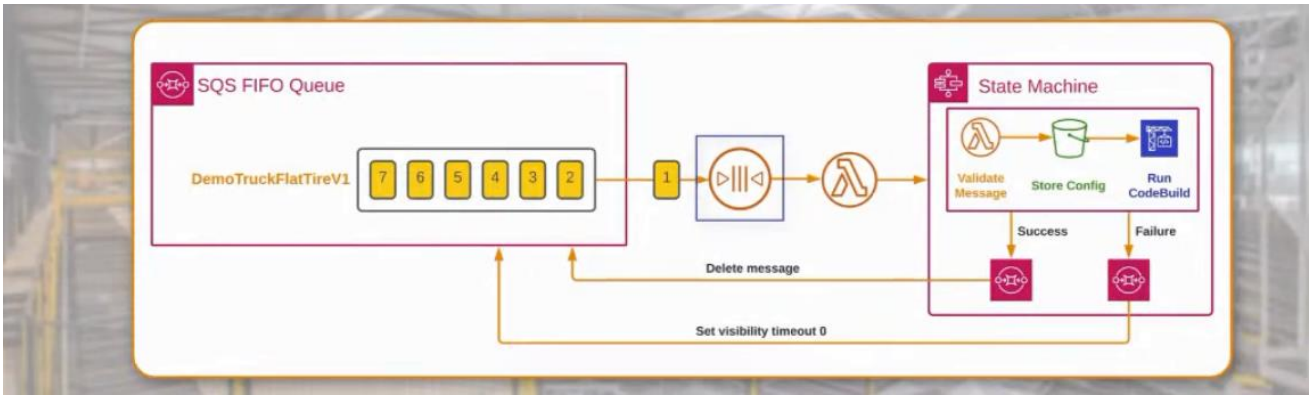
Only one operation can be performed on a single CF Stack at a time, so we need a method that can allow multiple operations to be executed on the same stack in order. We can use an SQS FIFO queue that are based on the concept called groupIds and messages placed on the same queue having the same groupIds are guaranteed to be processed in order. We can use the unique stack names to serve as the groupId value for this use-case.



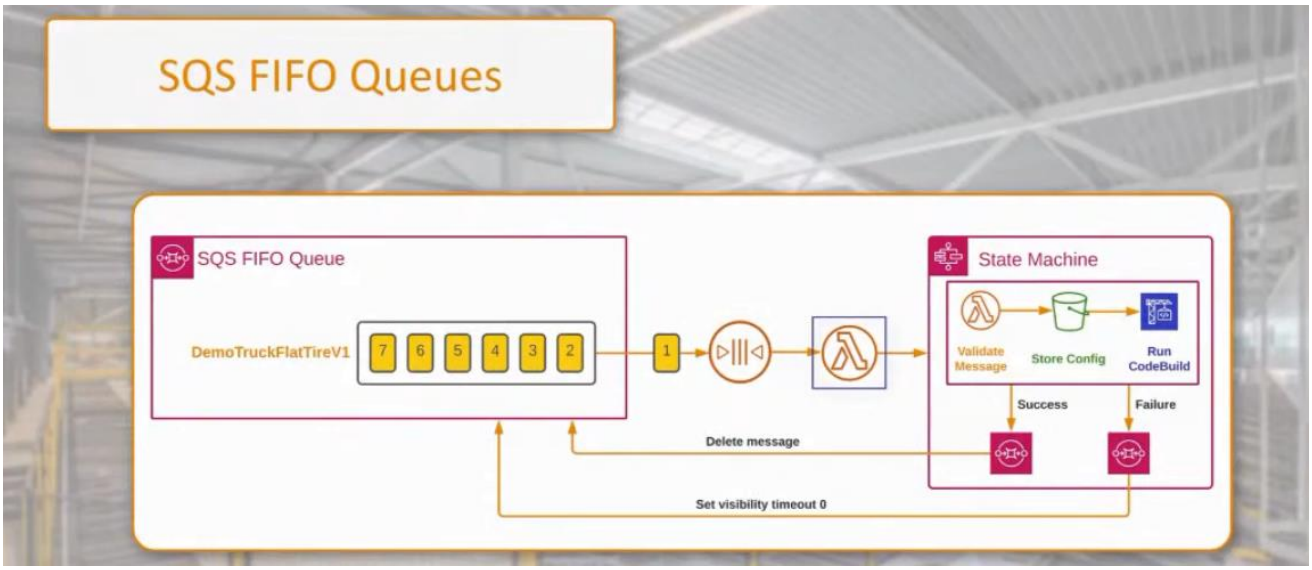
The last part is how to trigger CodeBuild from an SQS queue. Since SQS is a Pull-based mechanism and Step Functions is a Push-based mechanism, we need some sort of glue to connect the them together.



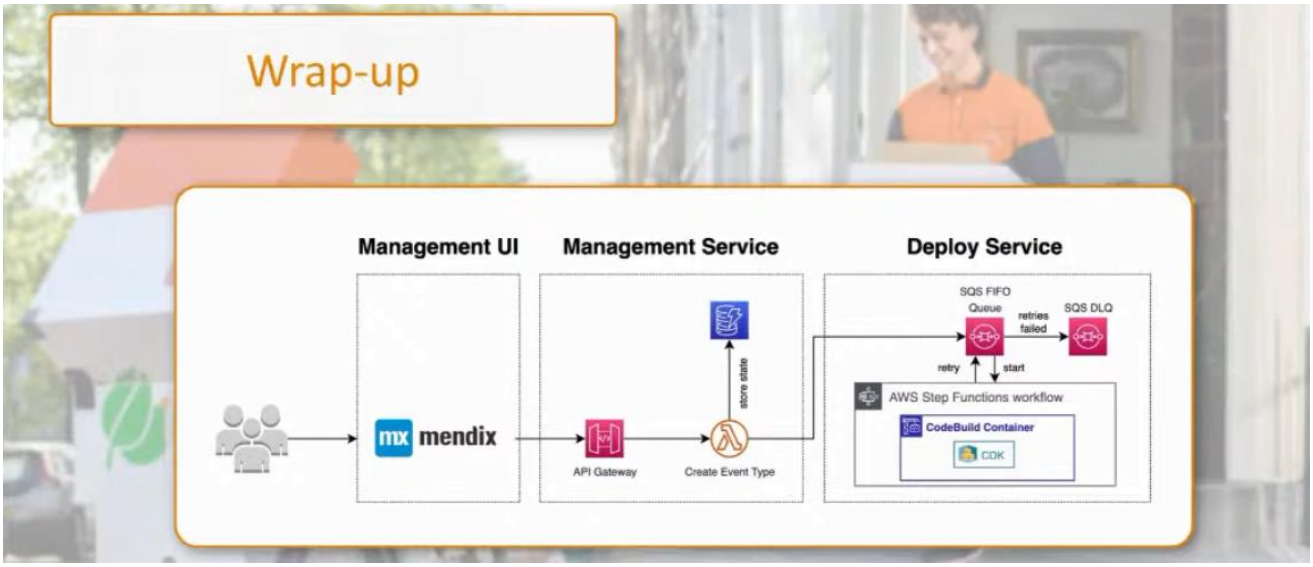
We can use a Lambda function with an event-source mapping as above.



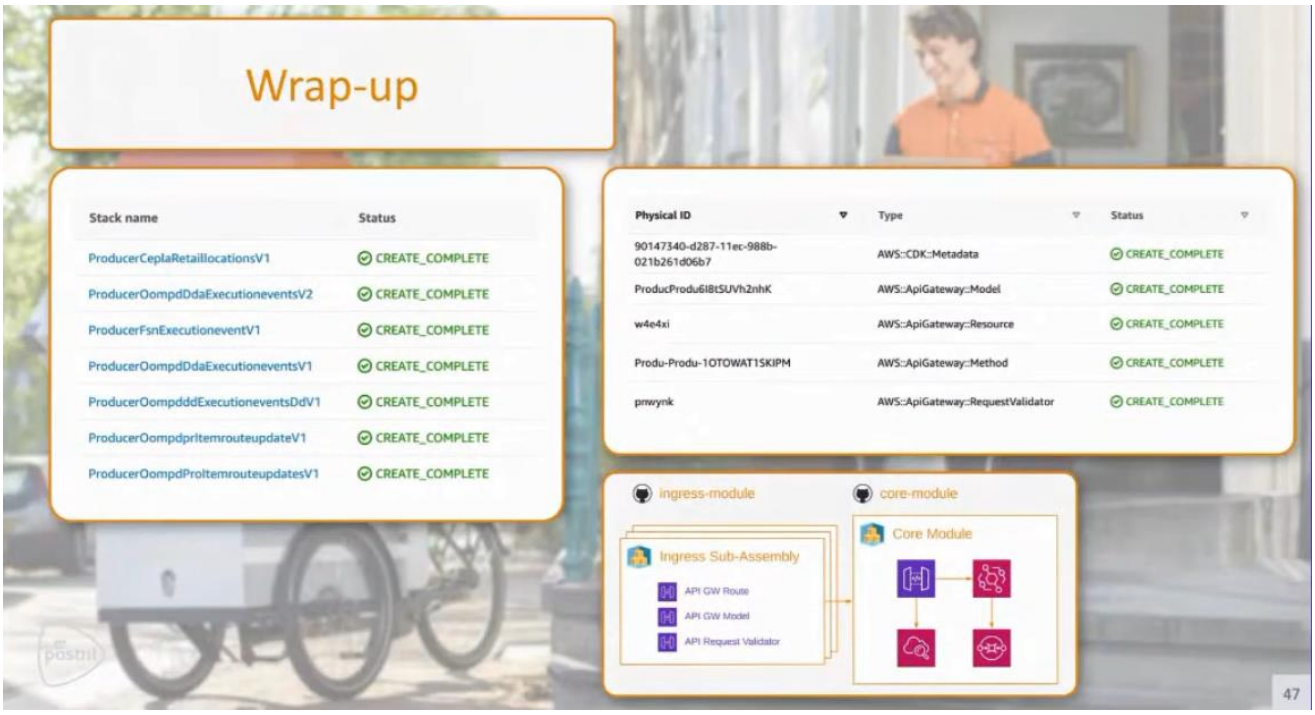
The event-source mapping continuously polls the queue and when new events arrive on any groupId, it invokes the Lambda function.



The Lambda function then forwards the message to the state machine/Step Function which runs our CodeBuild process. When all the steps in the state machine succeeds, the SQS message is deleted from the FIFO queue and the next operation can start. If a state machine step fails, the message is returned to the queue and its visibility timeout is set to zero so that it will be retried.



That is the entire flow.


Wrap-up
Stack name
Status
ProducerCeplaRetaillocationsV1
CREATE_COMPLETE
ProducerOompdDdaExecutioneventsV2
CREATE_COMPLETE
ProducerFsnExecutioneventV1
CREATE_COMPLETE
ProducerOompdDdaExecutioneventsV1
CREATE_COMPLETE
ProducerOompdddExecutioneventsDdV1
CREATE_COMPLETE
ProducerOompdprItemrouteupdateV1
CREATE_COMPLETE
ProducerOompdProItemrouteupdatesV1
CREATE_COMPLETE
Physical ID
Type
Status
90147340-d287-11ec-988b-021b261d06b7
AWS::CDK::Metadata
CREATE_COMPLETE
ProducProdu6I8tSUVh2nhK
AWS::ApiGateway::Model
CREATE_COMPLETE
w4e4xi
AWS::ApiGateway::Resource
CREATE_COMPLETE
Produ-Produ-1OTOWAT1SKIPM
AWS::ApiGateway::Method
CREATE_COMPLETE
pnwynk
AWS::ApiGateway::RequestValidator
CREATE_COMPLETE
ingress-module
core-module
Ingress Sub-Assembly
API GW Route
API GW Model
API Request Validator
Core Module
47