Architecture Overview:

The system will use the following AWS services:

* AWS Lambda for serverless compute
* Amazon DynamoDB for token storage
* Amazon SES for token delivery
* Amazon API Gateway for RESTful API endpoints
* AWS IAM for security and access control

Architecture Diagram:

[User] -> [API Gateway] -> [Lambda Functions]

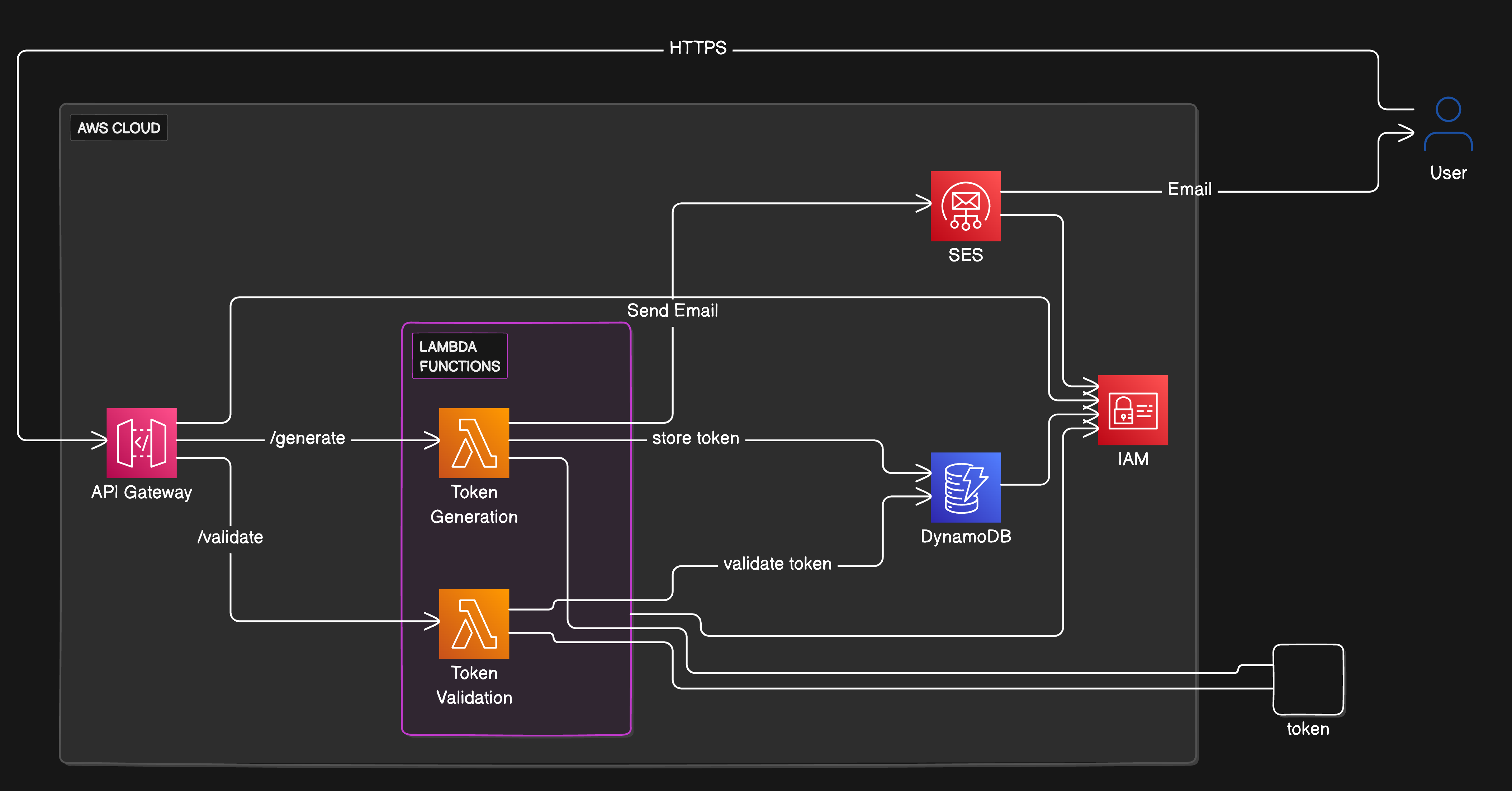
|

v

[DynamoDB] <-> [SES]

|

v

[CloudWatch]   


Detailed Components:

1. API Gateway:
   * Exposes two endpoints: /generate-token and /validate-token
   * Uses Lambda integration
2. Lambda Functions:  
   a. Token Generation Function:
   * Generates a token
   * Stores token in DynamoDB with expiration time
   * Sends token via SES

b. Token Validation Function:

* + Receives token from user
  + Checks token validity in DynamoDB
  + Deletes token if valid (single-use)

1. DynamoDB:
   * Store validation token in table.
2. SES:
   * Used to send SES email with tokens

Code Excerpts:

1. Token Generation Lambda (Python):

import json

import boto3

import time

from datetime import datetime, timedelta

import random

import jwt

import os

# Import Cognito Identity Provider client

dynamodb = boto3.resource('dynamodb')

# Get environment variables

SEND\_VERIFICATION\_CODE\_LAMBDA\_ARN = os.environ.get('SendVerificationCodeLambdaArn')

DYNAMODB\_TABLE\_NAME = os.environ.get('UserTokensTableName')

SENDER\_EMAIL = os.environ.get('SenderEmail')

SECRET\_KEY = os.environ.get('SecretKeyJwtToken')

# Create a Lambda client

lambda\_client = boto3.client('lambda')

def invoke\_email\_lambda(sender\_email, receiver\_email, subject, body):

payload = {

'sender\_email': sender\_email,

'receiver\_email': receiver\_email,

'subject': subject,

'body': body

}

response = lambda\_client.invoke(

FunctionName=SEND\_VERIFICATION\_CODE\_LAMBDA\_ARN,

InvocationType='Event', # Use 'RequestResponse' for synchronous execution, 'Event' for asynchronous

Payload=json.dumps(payload)

)

return response

def lambda\_handler(event, context):

table = dynamodb.Table(DYNAMODB\_TABLE\_NAME)

body = json.loads(event['body'])

# Access the 'email' attribute from the parsed data

email = body['email']

token = jwt.encode({'email': email, 'exp': datetime.utcnow() + timedelta(minutes=30)}, SECRET\_KEY, algorithm='HS256')

verification\_code = random.randint(10000, 999999)

table.put\_item(

Item={

'email': email,

'token': token,

'expiration\_time': int(time.time()) + 1800,

'verification\_code': verification\_code

}

)

# Send email to the specified email address

receiver\_email = email

email\_subject = "Verification Code"

email\_body = f"Your verification code is: {verification\_code}"

response = invoke\_email\_lambda(SENDER\_EMAIL, receiver\_email, email\_subject, email\_body)

return {

'statusCode': 200,

'body': json.dumps({

'message': 'Token created and saved in DynamoDB table "UserTokensTable"',

'verification\_code': verification\_code

})

}

1. Token Validation Lambda (Python):

import json

import jwt

import boto3

import time

from decimal import Decimal

from boto3.dynamodb.types import TypeDeserializer

import os

dynamodb = boto3.resource('dynamodb')

DYNAMODB\_TABLE\_NAME = os.environ.get('UserTokensTableName')

SECRET\_KEY = os.environ.get('SecretKeyJwtToken')

table = dynamodb.Table(DYNAMODB\_TABLE\_NAME)

deserializer = TypeDeserializer()

def get\_verification\_code\_item(verification\_code):

response = table.scan(

FilterExpression='verification\_code = :vc',

ExpressionAttributeValues={':vc': verification\_code}

)

return response['Items'][0] if response['Items'] else None

def deserialize\_item(item):

email = item['email']

token = item['token']

expiration\_time = deserializer.deserialize({'N': str(item['expiration\_time'])})

return email, token, expiration\_time

def decode\_token(token):

try:

decoded\_token = jwt.decode(token, SECRET\_KEY, algorithms=['HS256'])

return decoded\_token['email'], None

except jwt.ExpiredSignatureError:

return None, 'Token has expired'

except jwt.InvalidTokenError:

return None, 'Invalid token'

def verify\_token\_email(token\_email, item\_email):

return token\_email == item\_email

def is\_token\_expired(expiration\_time):

return int(time.time()) > expiration\_time

def response(status\_code, message):

return {

'statusCode': status\_code,

'body': json.dumps(message)

}

def lambda\_handler(event, context):

body = json.loads(event['body'])

verification\_code = int(body['verification\_code'])

item = get\_verification\_code\_item(verification\_code)

if not item:

return response(400, 'Invalid verification code')

email, token, expiration\_time = deserialize\_item(item)

token\_email, error\_message = decode\_token(token)

if error\_message:

return response(400, error\_message)

if not verify\_token\_email(token\_email, email):

return response(400, 'Email mismatch')

if is\_token\_expired(expiration\_time):

return response(400, 'Token has expired')

return response(200, {

'email': email,

'message': "Token verified successfully"

Security Best Practices:

1. Use IAM roles with least privilege for Lambda functions
2. Enable encryption at rest for DynamoDB
3. Use HTTPS for API Gateway
4. Implement rate limiting on API Gateway
5. Use AWS WAF to protect against common web exploits

Health/Availability & Performance Metrics:

1. CloudWatch Metrics to monitor:
   * Lambda invocation count and duration
   * API Gateway request count and latency
   * DynamoDB read/write capacity units
   * SNS message delivery rate
2. Use AWS X-Ray for tracing requests across services

Resilience and Modularity:

1. Use multi-AZ deployment for DynamoDB
2. Implement retry logic in Lambda functions
3. Use DynamoDB global tables for multi-region resilience
4. Separate Lambda functions by concern (generation vs. validation)

This design meets all the requirements:

* 100% built on AWS
* Serverless (using Lambda, DynamoDB, SNS)
* Resilient (multi-AZ, potential for multi-region)
* Modular (separate functions for generation and validation)