Serverless Event Announcement System

Neel Barvaliya

Introducing Today's Project!

Industry Relevance

This project mirrors real-world challenges that modern software companies face when building scalable, secure web applications. Event announcement systems are fundamental to platforms like Eventbrite, Facebook Events, Meetup, and corporate communication tools like Slack, where millions of notifications must be delivered reliably and securely.

The architectural patterns demonstrated here - serverless event-driven notifications, Infrastructure as Code, secure CDN deployment, and comprehensive security layers - are industry standards used by companies like Netflix, Airbnb, and Spotify to manage their cloud infrastructure.

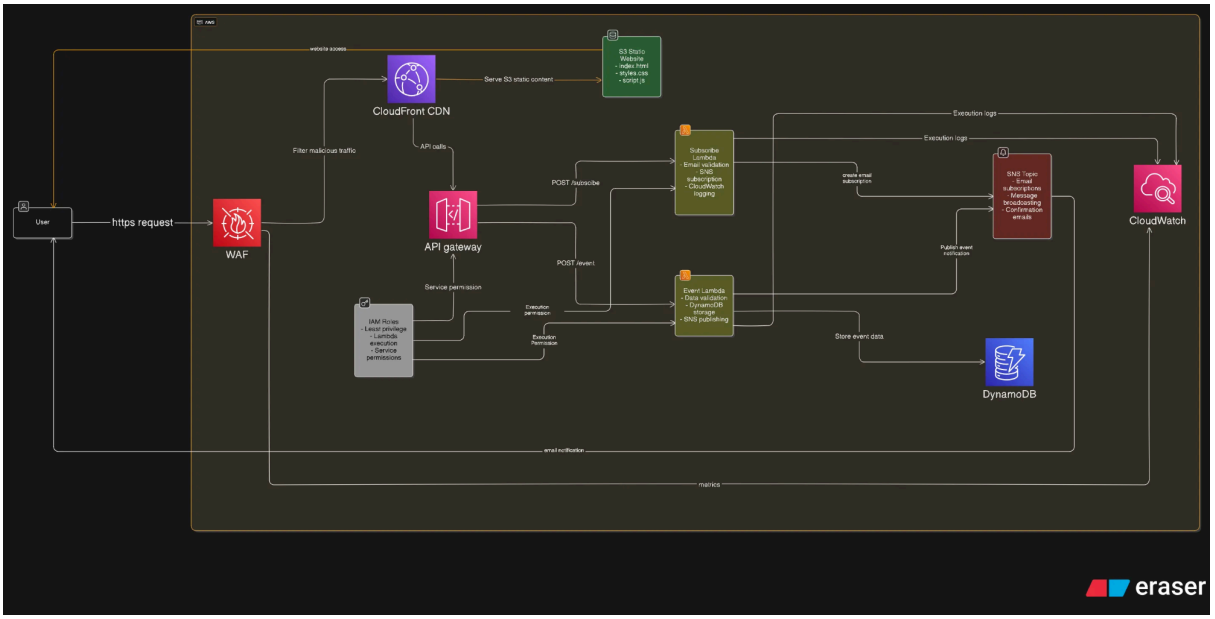
The specific challenges addressed (API rate limiting, HTTPS certificate management, WAF configuration, and CloudWatch monitoring) represent common production scenarios that cloud engineers encounter daily. Major tech companies rely heavily on AWS services like Lambda, SNS, CloudFront, and WAF for their core operations, making these skills directly transferable to enterprise environments. The emphasis on security-first deployment and infrastructure automation reflects current industry best practices driven by compliance requirements and the need for reliable, auditable deployments.

What are AWS Lambda, SNS, and CloudFront?

Amazon Lambda is a serverless compute service that runs code in response to events without provisioning servers. Amazon SNS (Simple Notification Service) is a messaging service that coordinates message delivery to multiple subscribers via email, SMS, or other protocols. Amazon CloudFront is a content delivery network (CDN) that caches content globally and provides HTTPS termination for web applications.

How I used AWS services in this project

In this project, I built a complete event announcement system with email subscriptions and real-time notifications. Users can subscribe via a web form hosted on S3, and when events are created through API Gateway endpoints, all subscribers receive immediate email notifications via SNS. The system uses Lambda for serverless processing, DynamoDB for event storage, CloudFront for secure global content distribution, and WAF for comprehensive security protection.



Email Subscription and Event Creation Flow

The core functionality implements a dual-endpoint system that handles both email subscriptions and event broadcasting through a secure, scalable architecture.

```
pwsh  DESKTOP-F4J15D2/Asus                                    29.493s · 04/09/25 12:50
[ » » » » » terraform ]
❯ curl -X POST https://2dtu2xkiw7.execute-api.us-east-1.amazonaws.com/dev/subscribe ` -H "Content-Type: application/json" ` -d '
{"email": "neilpwith123@gmail.com"}'
"Subscription initiated for neilpwith123@gmail.com. Check your email to confirm!"
```

Subscribe Lambda Function

```
import json
import boto3
import re
```

```
import logging

sns = boto3.client('sns')
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

def email_validation(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    isValid = re.match(pattern, email) is not None
    logger.debug(f"Validating email: {email}, Is valid: {isValid}")
    return isValid

def lambda_handler(event, context):
    logger.info("Source IP: " + event['requestContext']['identity']['sourceIp'])
    logger.info(f"Received event: {json.dumps(event['body'])}")

    try:
        body = json.loads(event['body'])
        email = body.get('email')

        if not email:
            logger.warning("No email provided in the request")
            return build_response(400, {'error':'No email provided!'})

        if not email_validation(email):
            logger.warning(f"Invalid email format: {email}")
            return build_response(400, {'error': 'Invalid email format, check agai
n!'})

        logger.info(f"Subscribing email: {email} to SNS topic")
        sns.subscribe(
            TopicArn=os.environ['SNS_TOPIC_ARN'],
            Protocol='email',
            Endpoint=email
        )

        logger.info(f"Subscription initiated for {email}")
        return build_response(200, f'Subscription initiated for {email}. Check y
```

```
our email to confirm!')

    except Exception as e:
        logger.error(f"Error subscribing email: {str(e)}", exc_info=True)
        return build_response(500, f'Error subscribing email: {str(e)}')
```

Infrastructure as Code

Using Terraform for AWS Infrastructure

I used Terraform to manage all AWS resources as code, including S3 buckets, Lambda functions, API Gateway, DynamoDB tables, CloudFront distribution, and WAF rules. This approach ensures reproducible infrastructure and follows DevOps best practices.

```
aws_api_gateway_stage.dev: Creation complete after 0s [id=ags-2dtu2xkiw7-dev]

Apply complete! Resources: 19 added, 0 changed, 0 destroyed.

Outputs:

api_gateway_event_url = "https://2dtu2xkiw7.execute-api.us-east-1.amazonaws.com/dev/event"
api_gateway_subscribe_url = "https://2dtu2xkiw7.execute-api.us-east-1.amazonaws.com/dev/subscribe"
```

SNS Topic and Lambda Integration

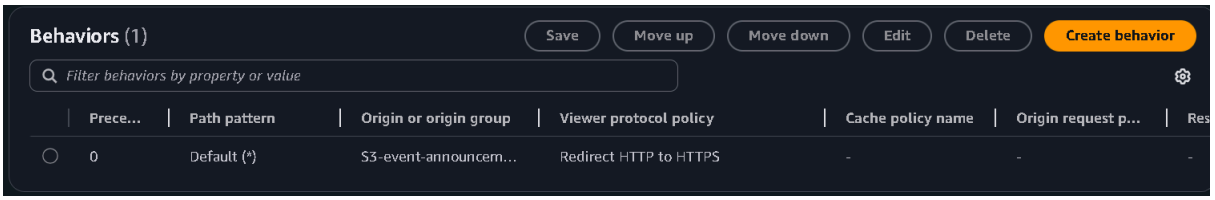
```
# SNS topic for email notifications
resource "aws_sns_topic" "event_announcement_topic" {
  name = "event-announcement-topic"

  tags = {
    Name = "EventAnnouncementTopic"
    Environment = "development"
  }
}

# Lambda function for event creation
resource "aws_lambda_function" "create_event" {
  filename = "create_event_lambda.zip"
  function_name = "create_event"
  role = aws_iam_role.lambda_event_role.arn
  handler = "create_event_lambda.lambda_handler"
```

```
  runtime = "python3.9"

  environment {
    variables = {
      SNS_TOPIC_ARN = aws_sns_topic.event_announcement_topic.arn,
      DYNAMODB_TABLE_EVENT = aws_dynamodb_table.events_table.name
    }
  }
}
```

Secure HTTPS Deployment with CloudFront

Origin Access Control Implementation

Traditional S3 static websites only support HTTP. I implemented CloudFront with Origin Access Control (OAC) to provide HTTPS termination and secure S3 access, replacing the deprecated Origin Access Identity approach.



Secure S3 Bucket Policy for OAC

```
resource "aws_s3_bucket_policy" "frontend_event_bucket_policy" {
  bucket = aws_s3_bucket.frontend_event_bucket.id

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Sid    = "AllowCloudFrontServicePrincipal"
        Effect = "Allow"
        Principal = {
          Service = "cloudfront.amazonaws.com"
        }
        Action   = "s3:GetObject"
```

```
      Resource = "${aws_s3_bucket.frontend_event_bucket.arn}/*"
      Condition = {
        StringEquals = {
          "AWS:SourceArn" = aws_cloudfront_distribution.event_announceme
nt_frontend_distribution.arn
        }
      }
    }
  ]
 })
}
```

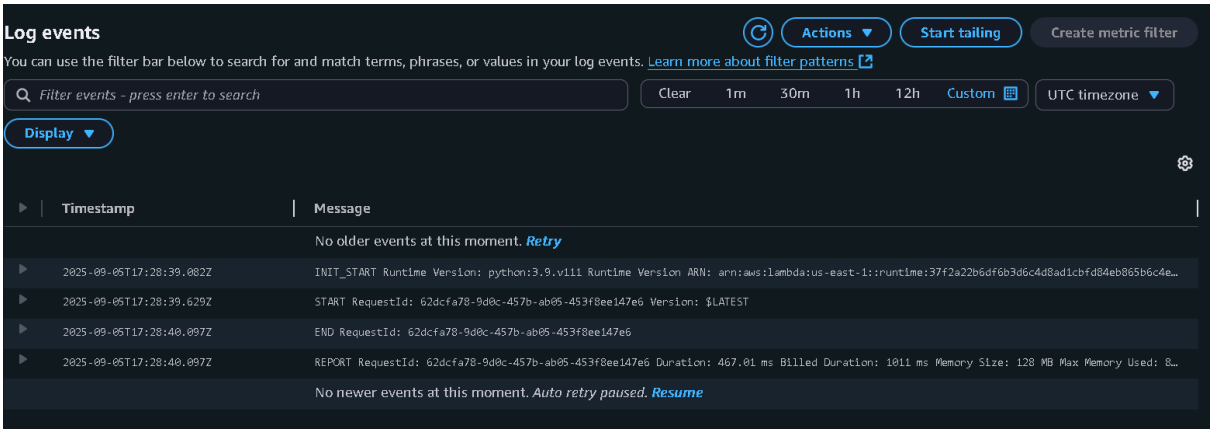
Web Application Firewall (WAF) Protection

Multi-Layer Security Implementation

I implemented AWS WAF to protect against common web attacks, including rate limiting, SQL injection, and cross-site scripting attempts. The WAF sits between users and CloudFront, filtering malicious requests before they reach the application.

Comprehensive CloudWatch Logging

Production-Ready Monitoring Implementation

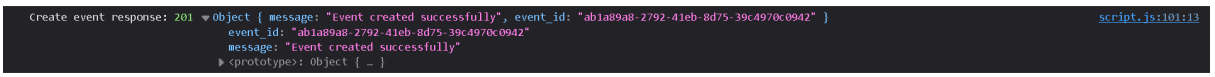


I implemented detailed CloudWatch logging across both Lambda functions to track user interactions, API calls, error conditions, and system performance. This provides full observability for debugging and audit purposes.

Event Creation Lambda with Logging

```
import boto3
import json
import uuid
import time
import logging
from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')
sns = boto3.client('sns')
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    logger.info("Source IP: " + event['requestContext']['identity']['sourceIp'])
    logger.info(f"Received event: {json.dumps(event['body'])}")

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

        # Validate required fields
        required_fields = ['event_title','event_datetime']
        for field in required_fields:
            if field not in body:
                logger.warning(f"{field}: is required for successful event creatio
n")
                return build_response(400, f'{field} is required!')

        # Create event data
        event_data = {
            'event_id': str(uuid.uuid4()),
            'event_title': body['event_title'][:50],
            'event_datetime': int(body['event_datetime']),
            'created_at': int(time.time()),
        }

        if body.get('event_description'):
```

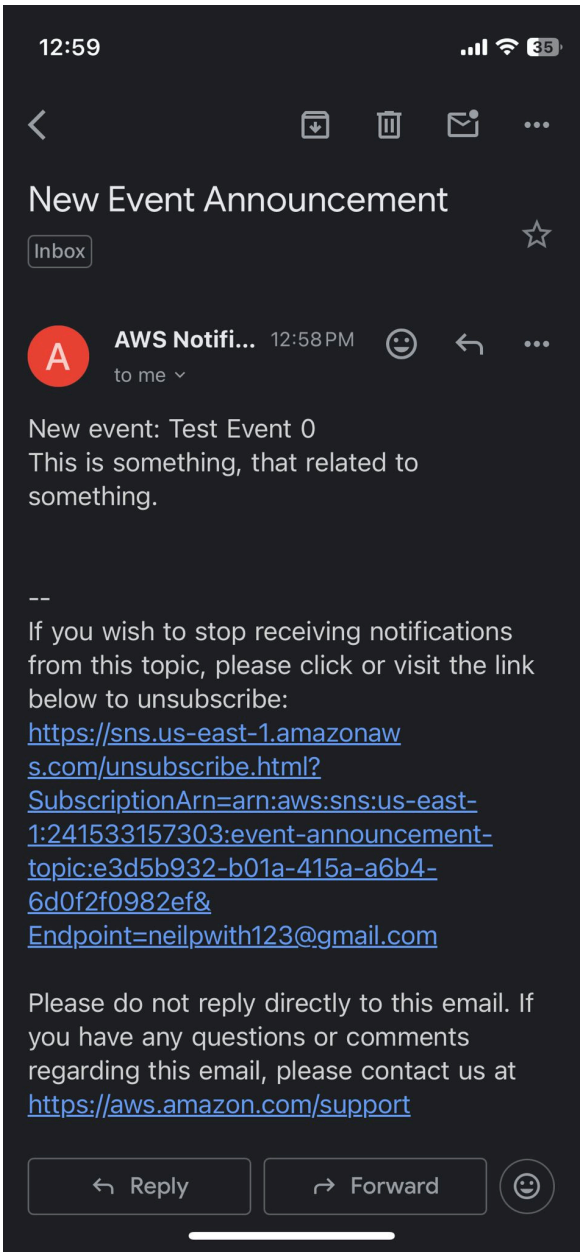
```
        event_data['event_description'] = body['event_description']
    if body.get('location'):
        event_data['location'] = body['location']

    logger.info(f"Saving event data: {event_data}")

    # Store in DynamoDB
    table = dynamodb.Table(os.environ['DYNAMODB_TABLE_EVENT'])
    table.put_item(Item=event_data)

    # Send SNS notification
    logger.info("Sending SNS notification")
    message = f"New event: {event_data['event_title']}\n"
    if event_data.get('event_description'):
        message += f"{event_data['event_description']}\n"
    if event_data.get('location'):
        message += f"Location: {event_data['location']}\n"

    sns.publish(
        TopicArn=os.environ['SNS_TOPIC_ARN'],
        Message=message,
        Subject='New Event Announcement'
    )

    logger.info("Event created successfully")
    return build_response(201, {
        'message': 'Event created successfully',
        'event_id': event_data['event_id']
    })

except Exception as e:
    logger.error(f"Error creating event: {str(e)}", exc_info=True)
    return build_response(500, f'Error creating event: {str(e)}')
```



Event-Driven Architecture

SNS Integration for Real-Time Notifications

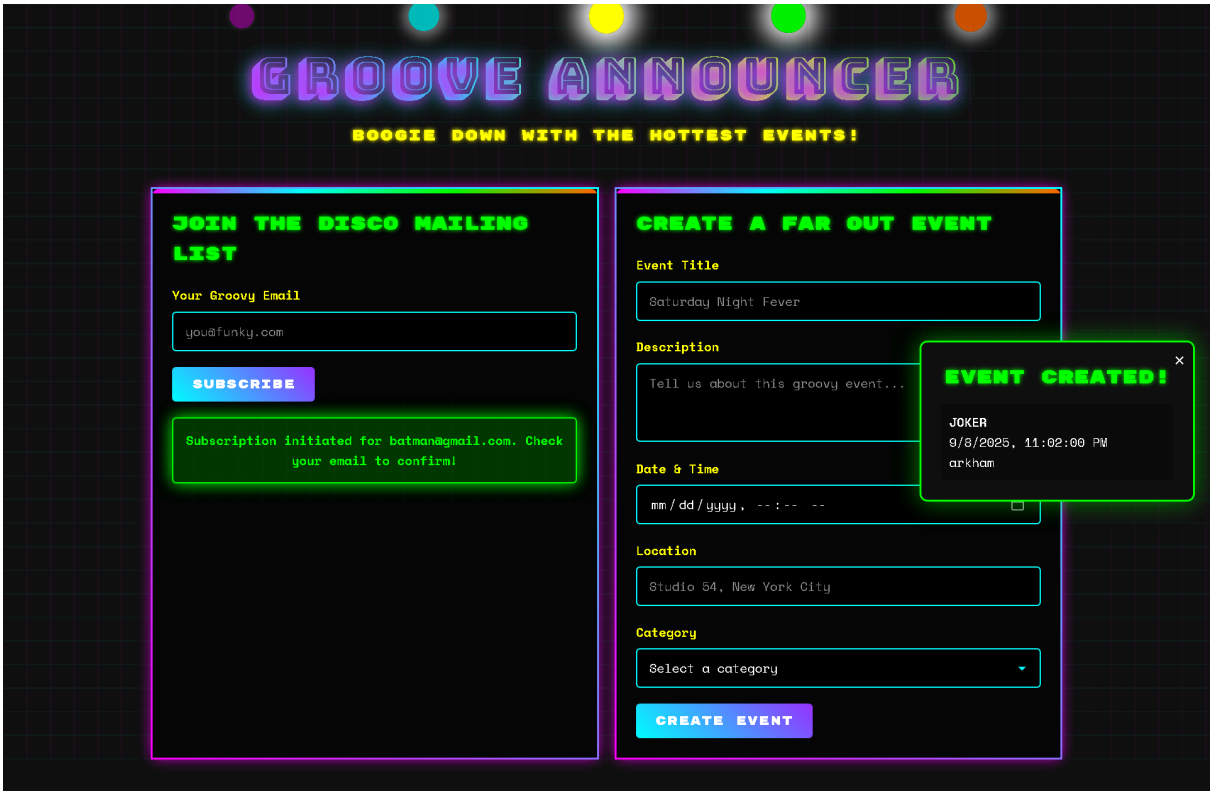
The system uses event-driven architecture where API Gateway triggers Lambda functions, which store data in DynamoDB and automatically broadcast notifications via SNS. This decouples the event creation process from notification delivery, making the system more resilient and scalable.



The notification flow ensures immediate delivery without requiring the API to handle email logic directly:

1. User creates event via frontend form
2. API Gateway triggers Event Creation Lambda

3. Lambda validates and stores event in DynamoDB
4. Lambda publishes message to SNS topic
5. SNS delivers emails to all confirmed subscribers
6. Users receive formatted event notifications



Real-World Skills Demonstration

This project demonstrates production-ready cloud engineering skills that directly translate to enterprise software development environments. The architecture patterns implemented here - serverless computing, event-driven design, Infrastructure as Code, and comprehensive security layers - are foundational to modern cloud applications.

The specific technical challenges overcome, including CloudWatch logging implementation, WAF rule configuration, CloudFront-S3 integration with OAC, and Terraform state management, represent real scenarios that cloud engineers face daily in production environments. The emphasis on security-first design, automated deployments, and comprehensive monitoring reflects industry best practices required for enterprise-grade applications.

The project showcases proficiency in AWS service integration, problem-solving under technical constraints, and the ability to research and implement evolving

cloud technologies. These skills are essential for roles in DevOps, cloud architecture, and full-stack development at technology companies.

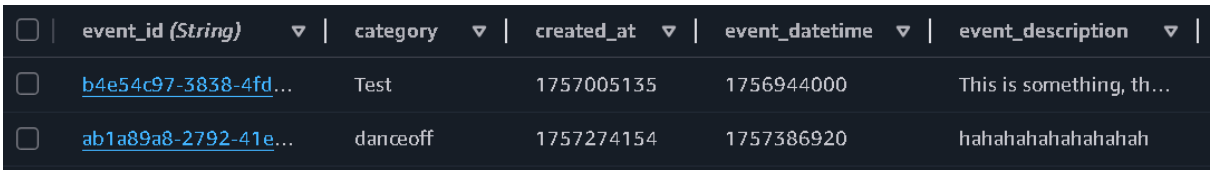
API Testing Results

Successful Event Creation Response

```
POST /event
{
  "message": "Event created successfully",
  "event_id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
}
```

Successful Subscription Response

```
POST /subscribe
{
  "message": "Subscription initiated for user@example.com. Check your e
mail to confirm!"
}
```



event_id (String)	category	created_at	event_datetime	event_description
b4e54c97-3838-4fd…	Test	1757005135	1756944000	This is something, th…
ab1a89a8-2792-41e…	danceoff	1757274154	1757386920	hahahahahahahahah

Cost vs. Complexity Analysis

Current System: Approximately $5-15/month for low-medium traffic

- Simple architecture, easy to maintain
- Suitable for startup or small business use cases

Enterprise Scale: $200-1000/month for high traffic

- Multi-region deployment with disaster recovery
- Dedicated email infrastructure and enhanced monitoring
- Requires DevOps team for maintenance

Hybrid Approach: $25-75/month for growing applications

- Single region with SQS decoupling
- ElastiCache for performance optimization
- Professional domain and enhanced security

The current architecture demonstrates solid foundational understanding while remaining cost-effective for learning and small-scale production use. The scaling strategies shown above represent real-world enterprise patterns used by companies processing millions of events daily.

The Final Result!

Live Deployment: https://d1pow7gt3b5s67.cloudfront.net/
Status: Fully operational with HTTPS, global CDN, and WAF protection

The event announcement system successfully demonstrates enterprise-grade patterns including Infrastructure as Code, secure deployment practices, event-driven architecture, and comprehensive security implementation.

Production-Ready Features:

- Automatic HTTPS redirect and security headers
- Comprehensive input validation and sanitization
- Dual-layer rate limiting (API Gateway + WAF)
- Origin Access Control for secure S3 access
- Environment variable-based configuration management
- Detailed audit logging for compliance and debugging

Scalability Considerations

The current architecture supports horizontal scaling through additional Lambda concurrent executions, DynamoDB auto-scaling, and CloudFront edge locations. For enterprise scale, the system could be enhanced with SQS queues for asynchronous processing, multiple availability zones, and database replication.

One thing I didn't expect in this project was...

The complexity of debugging multiple technical challenges that arose during development. I encountered several critical errors that required systematic

troubleshooting:

WAF Configuration Conflicts: Hit a major roadblock with WAF rule configuration where I had conflicting `override_action` and `action` statements in the same rule. The error message "You have used none or multiple values for a field that requires exactly one value" taught me that managed rule groups use `override_action` while custom rules use `action` statements.

Rate Limiting Architecture Gaps: Realized that having API keys in Terraform doesn't automatically enforce rate limiting - the Lambda functions need to validate API keys, or the rate limiting only applies when keys are provided.

Each challenge required researching current AWS best practices, reading updated documentation, and understanding the evolving cloud security landscape. This taught me the critical importance of staying current with AWS service updates and always testing security configurations thoroughly.

Project Repository: [GitHub Link]

Live Demo: https://d1pow7gt3b5s67.cloudfront.net/