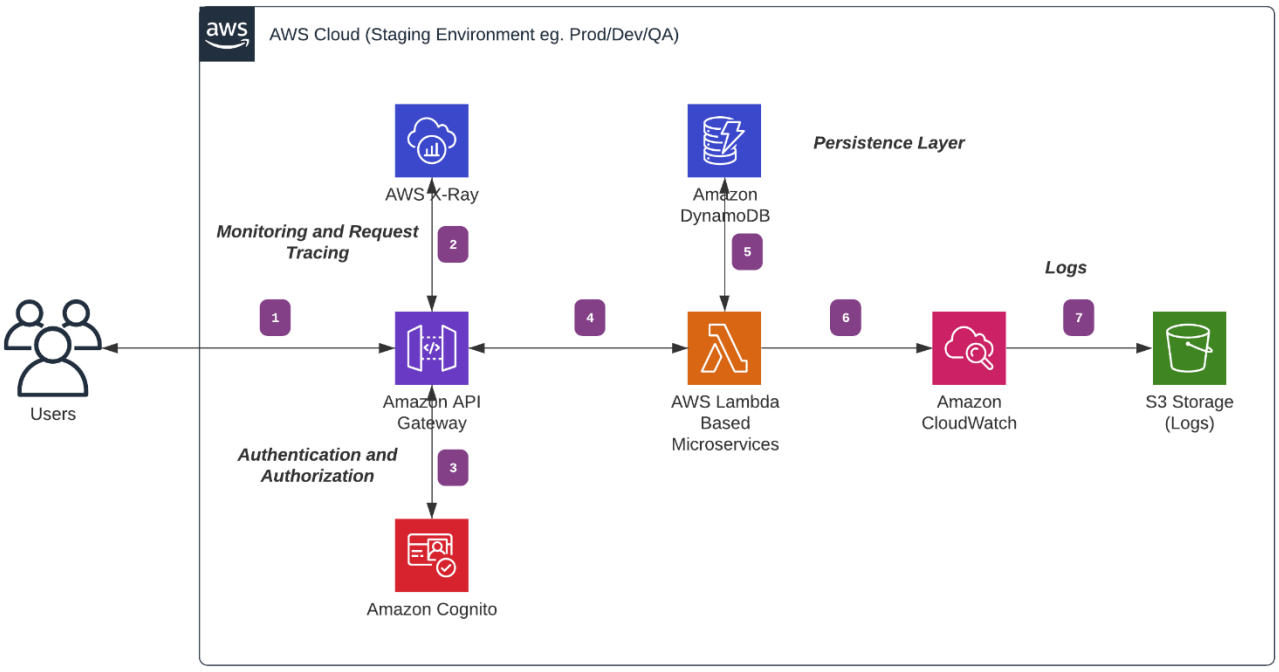
Scenario: Model and implement a web service platform using server-less technologies on a prominent IaaS provider.

1. Need to implement a CI/CD pipeline to deploy services to this platform. This CI/CD pipeline should be able to deploy the solution to multiple environments such as Development, Staging and Production.
2. Need to expose these services using an API layer and secure each and every service using a security token mechanism. Your solution should be able to deploy using a cloud deploy automation tool such as AWS CDK or using any suitable IaC tool.

Cloud Architecture

The following is the chosen solution for the above given problem description,



Evaluation and Justification

Evaluation and justification of the above architecture based on the Five pillars of AWS Well-Architecture Framework (AWS, 2021),

Operational excellence

- This CI/CD makes use of **Github** for the source repository, **Github Workflows** for pipelines, and the Infrastructure as Code toolset (**CDK** and **Terraform**).
- This delivers an easy-to-maintain continuous infrastructure and continuous delivery, lowering developer friction and increasing developer productivity.

Security

- **AuthN and AuthZ** - **AWS Cognito** offers a more complex version of the user onboarding journey, as well as more configurable options like Multifactor authentication. Amazon Cognito is also HIPAA compliant, as well as PCI DSS, SOC, and ISO/IEC 27001, ISO/IEC 27017, ISO/IEC 27018, and ISO 9001. AWS Cognito is used to safeguard all API endpoints in the **API Gateway**.
- **Encryption at Transit** - Because the resource endpoint is accessible through HTTPS, any man in the middle attacks is avoided.
- **Encryption at REST** - Aside from that, AWS encrypts DynamoDB data at **REST**. Because we maintain all of the PII information in Cognito, which passes the majority of the compliance tests, this adds an extra degree of protection.
- **Roles and Policies** – Each lambda is assigned a restricted policy that allows it to access only DynamoDB and CloudWatch. This restricts the use of any other AWS resources that aren't required for the lambda's operation.
- **Traceability** – The combination of **AWS CloudWatch**, **AWS CloudTrail**, and **AWS Xray** gives you more information about who made what modifications and how the resources were executed.

Reliability

- **Observability**: **AWS CloudWatch** allows you to save all of your lambda execution logs to a specific S3 bucket. **AWS XRay**, on the other hand, provides more information about how internal services receive traffic as well as an informative dashboard on executions.
- The entire architecture can be scaled horizontally to increase the aggregated system availability and an identical architecture can be spun up in matter of minutes with **no downtime**.

Performance

- Because the design relies on the serverless approach as its foundation, scaling up and down takes only a few minutes.
- The usage of **AWS API Gateway** provides a single endpoint for multiple microservices and enables additional capabilities such separate staging and versioning. Since **AWS API Gateway** is a managed service by AWS it guarantees scalability and High Availability.
- **AWS Lambda** was chosen instead on *AWS EC2*, *AWS ECR* or *AWS EKS* since all the aforementioned services needs to be managed manually and requires expertise to manage it. Moreover, with the use of AWS Lambda the application can be scaled on functional level.
- **AWS DynamoDB** is a fully managed NoSQL service provided by AWS. It also guarantees scalability and High Availability. In addition, if required, **DynamoDB table** could also be used to manage the IaC state file locks so that two different developers don't provision or change the same AWS service.
- The use of **AWS S3** to store logs, which can hold up to petabytes of data and can be migrated to a new tier after a specified time limit or deleted based on business requirements. S3 is also utilized to keep Terraform and CDK's state files up to date for change detection.

Cost Optimization

- **Consumption Modal** - The architecture tries to keep the **pay-for-usage** approach as much as possible. This assures that there will be no long running instances and the cost is only incurred based on consumption.
- **Reduce Cost of Ownership** – There is no need for specific engineers or data center operators because there are no managed services.

Appendix B: CDK (attached iac-cdk.zip)

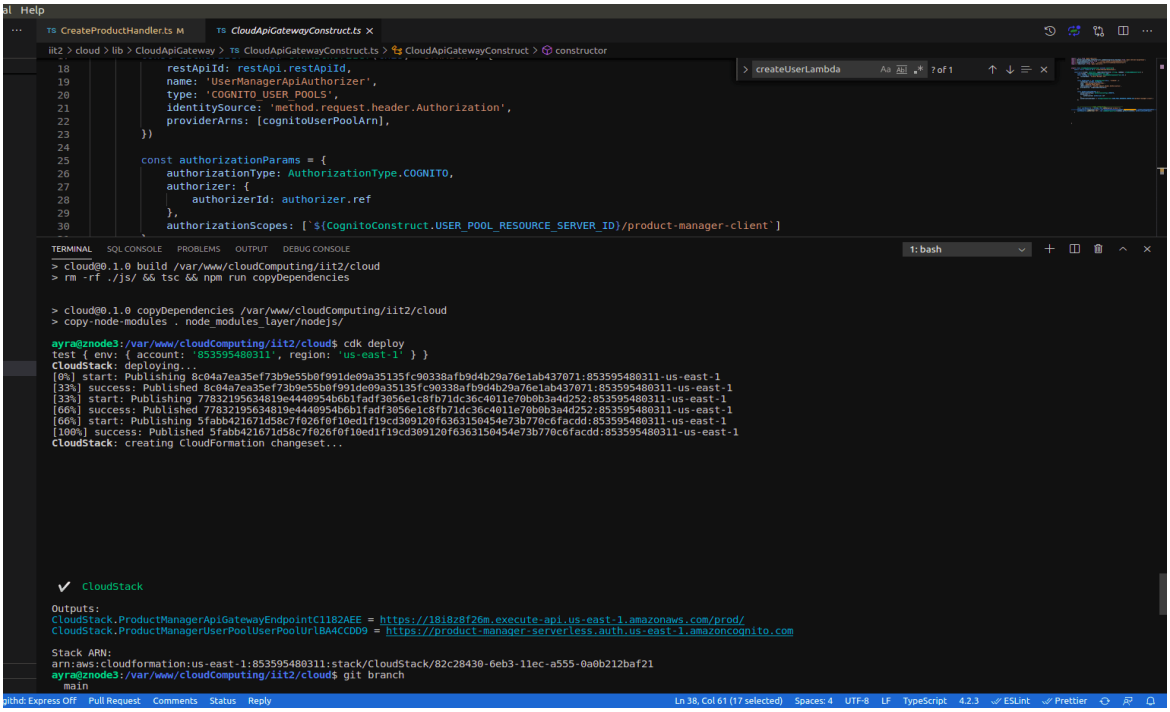


Figure 1 Screenshot of Deploying the infrastructure using AWS CDK

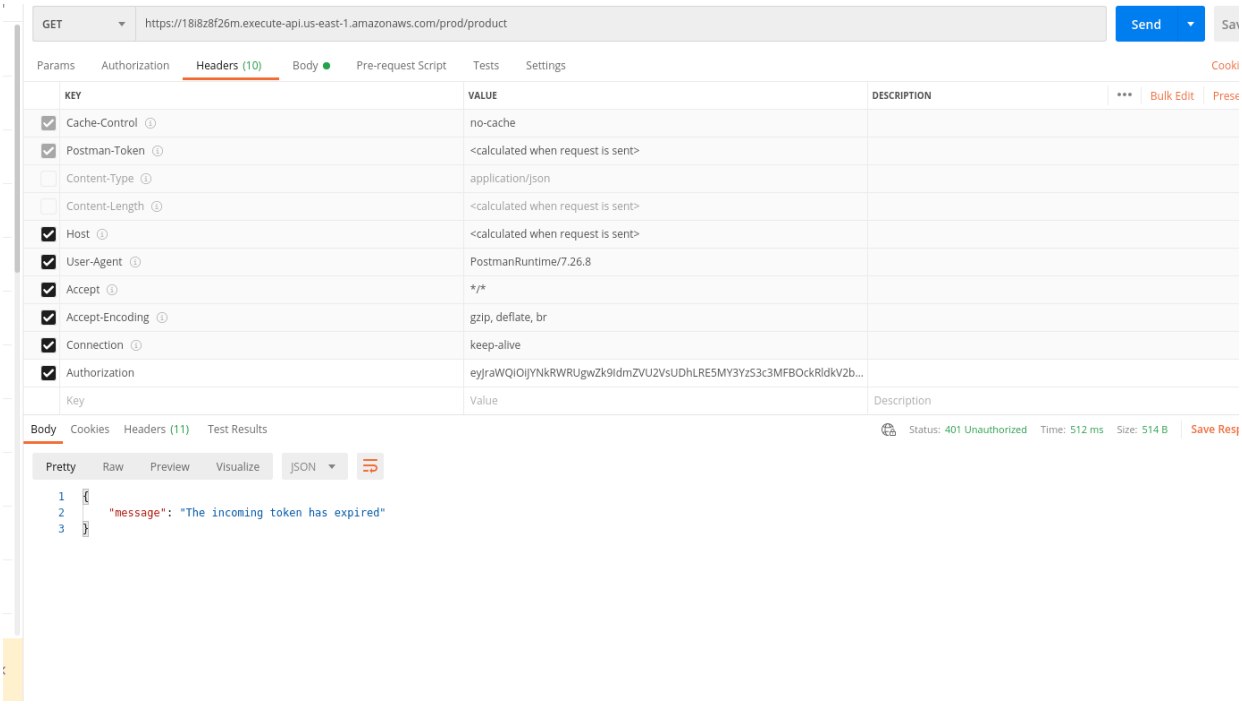


Figure 2 Screenshot of API Access with Invalid Access Token on Postman

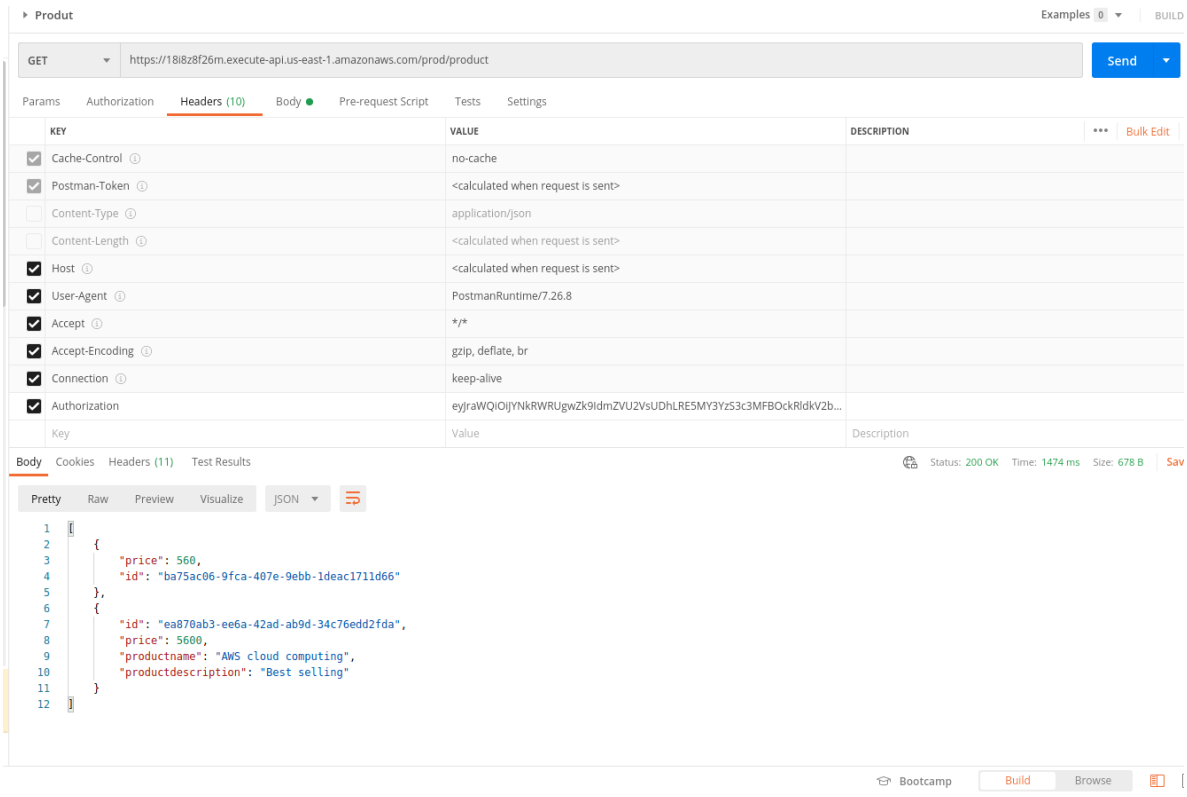


Figure 3 Screenshot of API Access with Valid Access Token on Postman

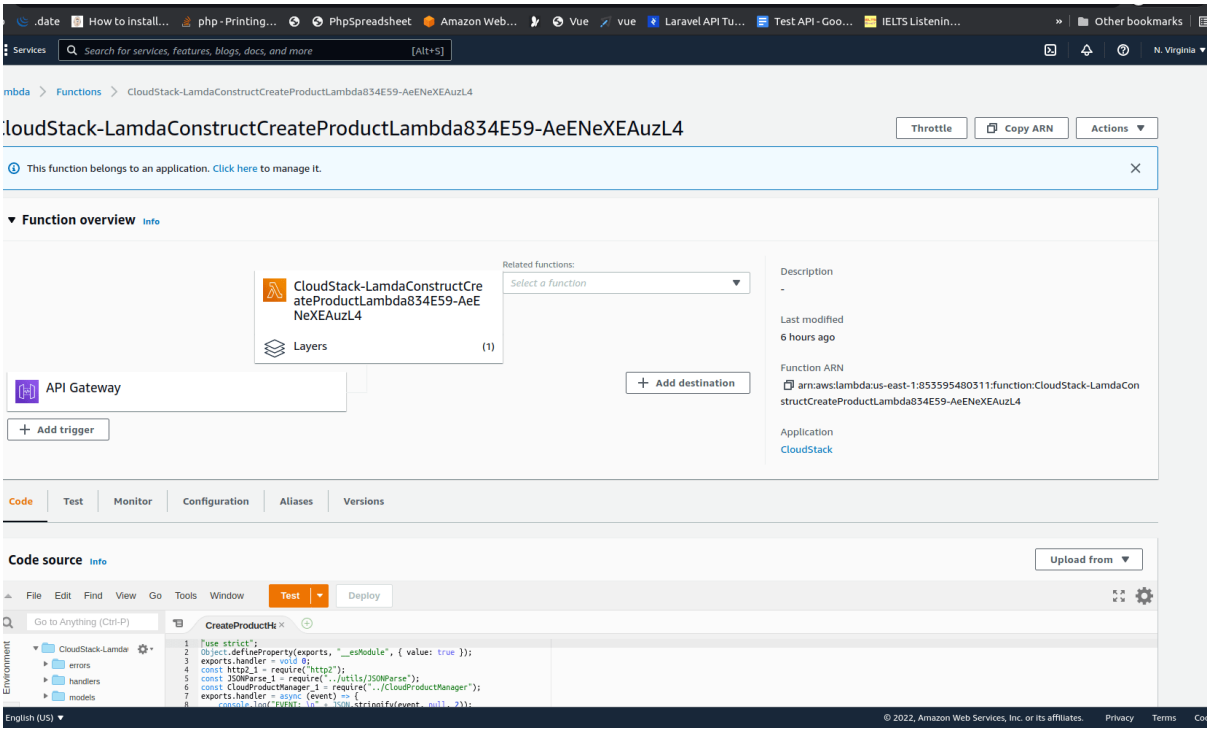


Figure 4 Screenshot of Create Product Lambda

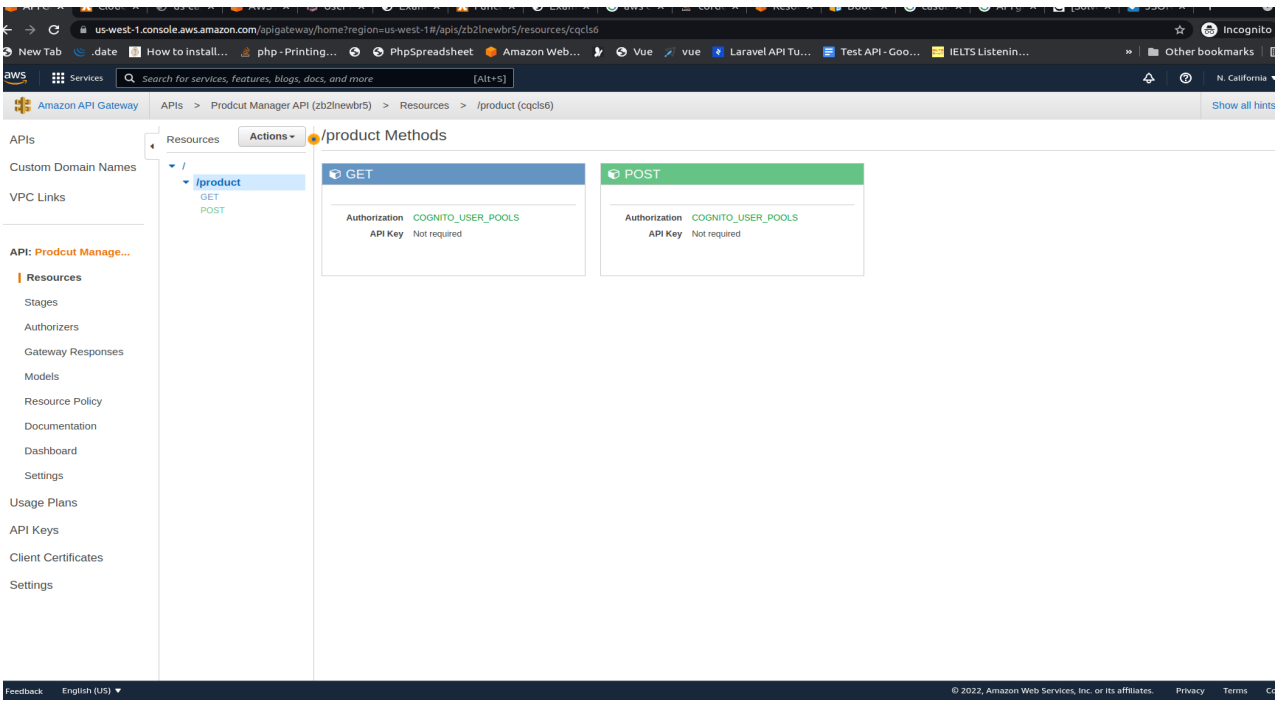


Figure 5 Screenshot of API Gateway

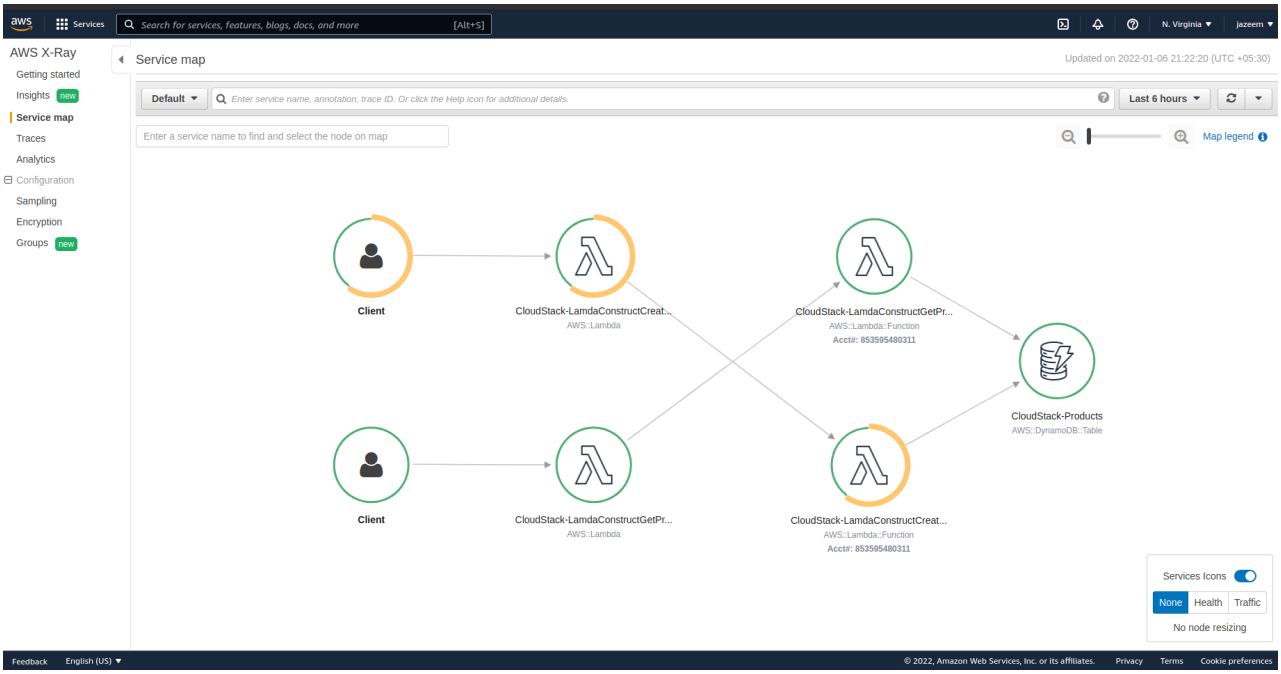
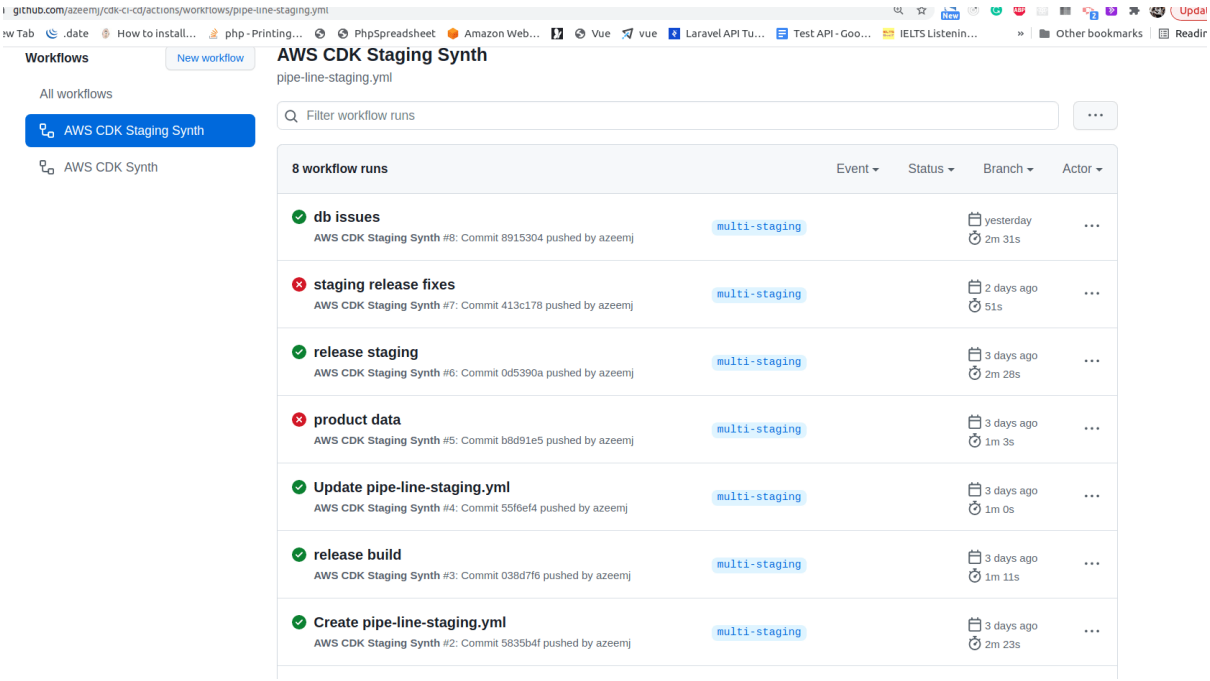


Figure 6 Screenshot of AWS Xray


github.com/azeemj/cdk-ci-cd/actions/workflows/pipe-line-staging.yml
Workflows
New workflow
All workflows
AWS CDK Staging Synth
AWS CDK Synth
AWS CDK Staging Synth
pipe-line-staging.yml
Filter workflow runs
8 workflow runs
Event
Status
Branch
Actor
db issues
AWS CDK Staging Synth #8: Commit 8915304 pushed by azeemj
multi-staging
yesterday
2m 31s
staging release fixes
AWS CDK Staging Synth #7: Commit 413c178 pushed by azeemj
multi-staging
2 days ago
51s
release staging
AWS CDK Staging Synth #6: Commit 0d5390a pushed by azeemj
multi-staging
3 days ago
2m 28s
product data
AWS CDK Staging Synth #5: Commit b8d91e5 pushed by azeemj
multi-staging
3 days ago
1m 3s
Update pipe-line-staging.yml
AWS CDK Staging Synth #4: Commit 55f6ef4 pushed by azeemj
multi-staging
3 days ago
1m 0s
release build
AWS CDK Staging Synth #3: Commit 038d7f6 pushed by azeemj
multi-staging
3 days ago
1m 11s
Create pipe-line-staging.yml
AWS CDK Staging Synth #2: Commit 5835b4f pushed by azeemj
multi-staging
3 days ago
2m 23s