AWS Bookstore Demo App

AWS Bookstore Demo App is a full-stack sample web application that creates a storefront (and backend) for customers to shop for fictitious books. The entire application can be created with a single **Terraform** template.

Overview

The goal of AWS Bookstore Demo App is to provide a fully-functional web application that utilizes multiple purpose-built AWS databases and native AWS components like Amazon API Gateway and AWS CodePipeline. Increasingly, modern web apps are built using a multitude of different databases. Developers break their large applications into individual components and select the best database for each job. Let's consider AWS Bookstore Demo App as an example. The app contains multiple experiences such a shopping cart, product search, recommendations, and a top sellers list. For each of these use cases, the app makes use of a purpose-built database so the developer never has to compromise on functionality, performance, or scale.

The provided Terraform template automates the entire creation and deployment of AWS Bookstore Demo App. The template includes the following components:

Database components

- **Product catalog/shopping cart** - *Amazon DynamoDB* offers fast, predictable performance for the key-value lookups needed in the product catalog, as well as the shopping cart and order history. In this implementation, we have unique identifiers, titles, descriptions, quantities, locations, and price.
- **Search** - *Amazon Elasticsearch* Service enables full-text search for our storefront, enabling users to find products based on a variety of terms including author, title, and category.
- **Recommendations** - *Amazon Neptune* provides social recommendations based on what user's friends have purchased, scaling as the storefront grows with more products, pages, and users.
- **Top sellers list** - *Amazon ElastiCache for Redis* reads order information from Amazon DynamoDB Streams, creating a leaderboard of the “Top 20” purchased or rated books.

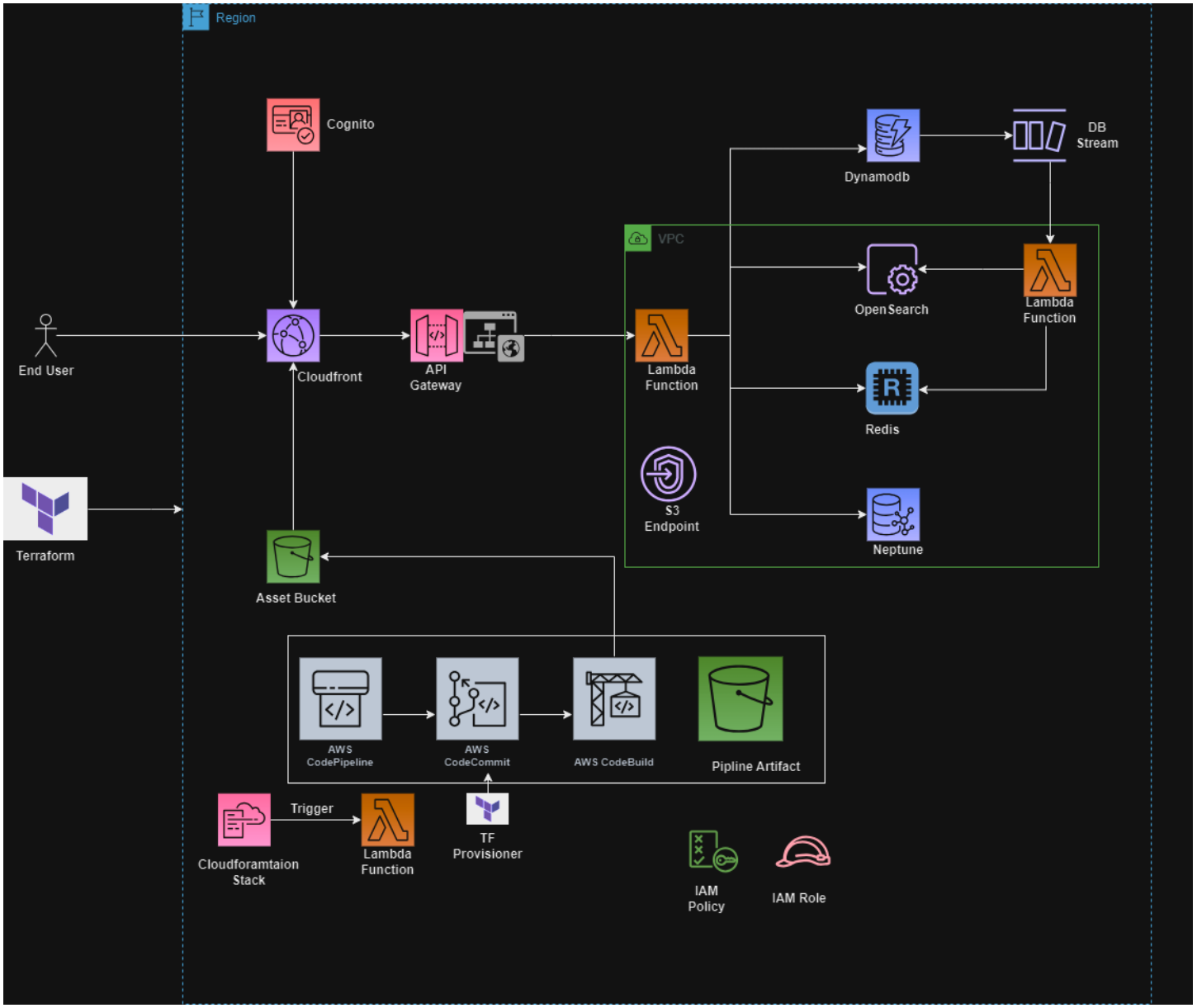
Application components

- **Serverless service backend** – *Amazon API Gateway* powers the interface layer between the frontend and backend, and invokes serverless compute with AWS Lambda.
- **Web application blueprint** – We include a React web application pre-integrated out-of-the-box with tools such as React Bootstrap, Redux, React Router, internationalization, and more.

Infrastructure components

- **Continuous deployment code** pipeline – *AWS CodePipeline* and *AWS CodeBuild* help you build, test, and release your application code.
- **Serverless web application** – *Amazon CloudFront* and *Amazon S3* provide a globally-distributed application.
-

Architecture



Frontend

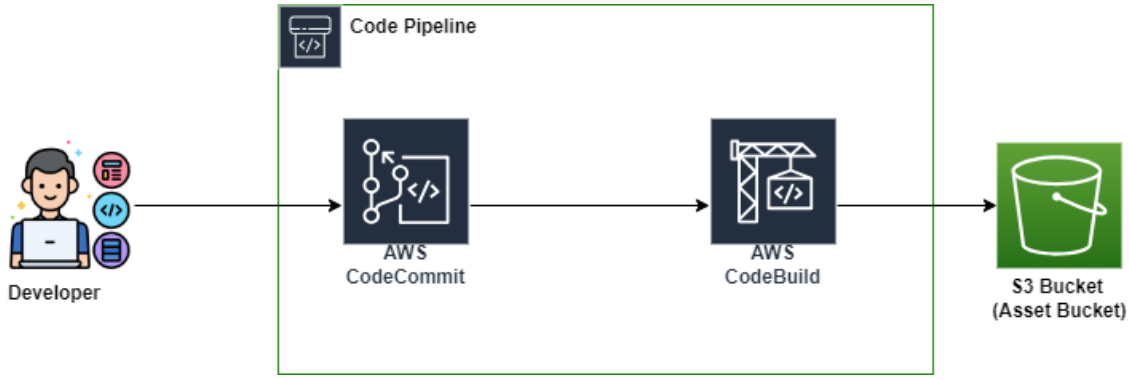
Build artifacts are stored in a S3 bucket where web application assets are maintained (like book cover photos, web graphics, etc.). Amazon CloudFront caches the frontend content from S3, presenting the application to the user via a CloudFront distribution. The frontend interacts with Amazon Cognito and Amazon API Gateway only. Amazon Cognito is used for all authentication requests, whereas API Gateway (and Lambda) is used for all API calls interacting across DynamoDB, Elasticsearch, ElastiCache, and Neptune.

Backend

The core of the backend infrastructure consists of Amazon Cognito, Amazon DynamoDB, AWS Lambda, and Amazon API Gateway. The application leverages Amazon Cognito for user authentication, and Amazon DynamoDB to store all of the data for books, orders, and the checkout cart. As books and orders are added, Amazon DynamoDB Streams push updates to AWS Lambda functions that update the Amazon Elasticsearch cluster and Amazon ElasticCache for Redis cluster. Amazon Elasticsearch powers search functionality for books, and Amazon Neptune stores information on a user's social graph and book purchases to power recommendations. Amazon ElasticCache for Redis powers the books leaderboard.

Developer Tools

The code is hosted in AWS CodeCommit. AWS CodePipeline builds the web application using AWS CodeBuild. After successfully building, CodeBuild copies the build artifacts into a S3 bucket where the web application assets are maintained (like book cover photos, web graphics, etc.). Along with uploading to Amazon S3, CodeBuild invalidates the cache so users always see the latest experience when accessing the storefront through the Amazon CloudFront distribution. AWS CodeCommit. AWS CodePipeline, and AWS CodeBuild are used in the deployment and update processes only, not while the application is in a steady-state of use.



Implementation details

Variables used in Terraform :

Set below variables as per your requirement in *terraform.tfvars* file:

- **account_id**
- **Region**
- **AssetsBucket** *# name for asset bucket (used to store the application files).*
- **Pipelinebucket** *# name for pipeline bucket (used to store the CodePipeline artifacts).*
- **ProjectName** *# Name for CodeBuild*
- **mybookstore-WebAssets** *# name for CodeCommit Repo.*
- **mybookstore-domain** *# Name for OpenSearch Domain*

Amazon DynamoDB

The backend of AWS Bookstore Demo App leverages Amazon DynamoDB to enable dynamic scaling and the ability to add features as we rapidly improve our e-commerce application. The application create three tables in DynamoDB: Books, Orders, and Cart. DynamoDB's primary key consists of a partition (hash) key and an optional sort (range) key. The primary key (partition and sort key together) must be unique.

- **Books Table:** Store the books details.
- **Orders Table:** Store order details.
- **Cart Table:** Store the cart related details (Items in cart)

Amazon API Gateway

Amazon API Gateway acts as the interface layer between the frontend (Amazon CloudFront, Amazon S3) and AWS Lambda, which calls the backend (databases, etc.). Below are the different APIs the application uses:

Books (DynamoDB)

GET /books (ListBooks)
GET /books/{:id} (GetBook)

Cart (DynamoDB)

GET /cart (ListItemsInCart)
POST /cart (AddToCart)
PUT /cart (UpdateCart)
DELETE /cart (RemoveFromCart)
GET /cart/{:bookId} (GetCartItem)

Orders (DynamoDB)

GET /orders (ListOrders)
POST /orders (Checkout)

Best Sellers (ElastiCache)

GET /bestsellers (GetBestSellers)

Recommendations (Neptune)

GET /recommendations (GetRecommendations)
GET /recommendations/{bookId} (GetRecommendationsByBook)

Search (OpenSearch)

GET /search (SearchES)

AWS Lambda

AWS Lambda is used in a few different places to run the application, as shown in the architecture diagram.. In the cases where the response fields are blank, the application will return a status Code 200 or 500 for success or failure, respectively.

- **ListBooks** Lambda function that lists the books in the specified product category
- **GetBook** Lambda function that will return the properties of a book.
- **ListItemsInCart** Lambda function that lists the items that user added in cart.
- **AddToCart** Lambda function that adds a specified book to the user's cart. Price is included in this function's request so that the price is passed into the cart table in DynamoDB.
- **RemoveFromCart** Lambda function that removes a given book from the user's cart.
- **GetCartItem** Lambda function that returns the details of a given item in user's cart.
- **UpdateCart** Lambda function that updates the user's cart with a new quantity of a given book.
- **ListOrders** Lambda function that lists the orders for a user.
- **Checkout** Lambda function that moves the contents of a user's cart (the books) into the checkout flow, where you can then integrate with payment, etc.

In addition to the above, the Checkout Lambda function acts as a sort of mini-workflow with the following tasks:

1. Add all books from the Cart table to the Orders table
2. Remove all entries from the Cart table for the requested customer ID

- **GetBestSellers** Lambda function that returns a list of the best-sellers.
- **GetRecommendations** Lambda function that returns a list of recommended books based on the purchase history of a user's friends.
- **GetRecommendationsByBook** Lambda function that returns a list of friends who have purchased this book as well as the total number of times it was purchased by those friends.

Other Lambda functions There are a few other Lambda functions used to make AWS Bookstore Demo App work, and they are listed here:

- **Search** - Lambda function that returns a list of books based on provided search parameters in the request.
- **UpdateSearchCluster** - Lambda function that updates the Elasticsearch cluster when new books are added to the store.
- **UpdateBestsellers** - Updates Leaderboard via the ElastiCache for Redis cluster as orders are placed.

Amazon ElastiCache for Redis

Amazon ElastiCache for Redis is used to provide the best sellers/leaderboard functionality. In other words, the books that are the most ordered will be shown dynamically at the top of the best sellers list.

For the purposes of creating the leaderboard, AWS Bookstore Demo App utilized ZINCRBY, which *"Increments the score of member in the sorted set stored at key by increment.* If member does not exist in the sorted set, it is added with increment

as its score (as if its previous score was 0.0). If key does not exist, a new sorted set with the specified member as its sole member is created."

The information to populate the leader board is provided from DynamoDB via DynamoDB Streams. Whenever an order is placed (and subsequently created in the **Orders** table), this is streamed to Lambda, which updates the cache in ElastiCache for Redis. The Lambda function used to pass this information is **UpdateBestSellers**.

Amazon Neptune

Neptune provides a social graph that consists of users, books. Recommendations are only provided for books that have been purchased (i.e. in the list of orders). The "top 5" book recommendations are shown on the bookstore homepage.

Amazon Elasticsearch

Amazon Elasticsearch Service powers the search capability in the bookstore web application, available towards the top of every screen in a search bar. Users can search by title, author, and category. The template creates a search domain in the Elasticsearch service.

It is important that a service-linked role is created first (Terraform configuration file)

AWS IAM

ListBooksLambda AWSLambdaBasicExecutionRole
dynamodb:Scan - table/Books/index/category-index
dynamodb:Query - table/Books

GetBookLambda AWSLambdaBasicExecutionRole
dynamodb:GetItem - table/Books

ListItemsInCartLambda AWSLambdaBasicExecutionRole
dynamodb:Query - table/Cart

AddToCartLambda AWSLambdaBasicExecutionRole
dynamodb:PutItem - table/Cart

UpdateCartLambda AWSLambdaBasicExecutionRole
dynamodb:UpdateItem - table/Cart

ListOrdersLambda AWSLambdaBasicExecutionRole
dynamodb:Query - table/Orders

CheckoutLambda AWSLambdaBasicExecutionRole
dynamodb:PutItem - table/Orders
dynamoDB:DeleteItem - table/Cart

Amazon Cognito

Amazon Cognito handles user account creation and login for the bookstore application. For the purposes of the demo, the bookstore is only available to browse after login, which could represent the architecture of different types of web apps. Users can also choose to separate the architecture, where portions of the web app are publicly available and others are available upon login.

User Authentication

- Email address

Amazon Cognito passes the CognitoIdentityID (which AWS Bookstore Demo app uses as the Customer ID) for every user along with every request from Amazon API Gateway to Lambda, which helps the services authenticate against which user is doing what.

Amazon CloudFront and Amazon S3

Amazon CloudFront hosts the web application frontend that users interface with. This includes web assets like pages and images. For demo purposes, CloudFormation pulls these resources from S3.

Amazon VPC

Amazon VPC (Virtual Private Cloud) is used with Amazon Elasticsearch Service, Amazon ElastiCache for Redis, and Amazon Neptune.

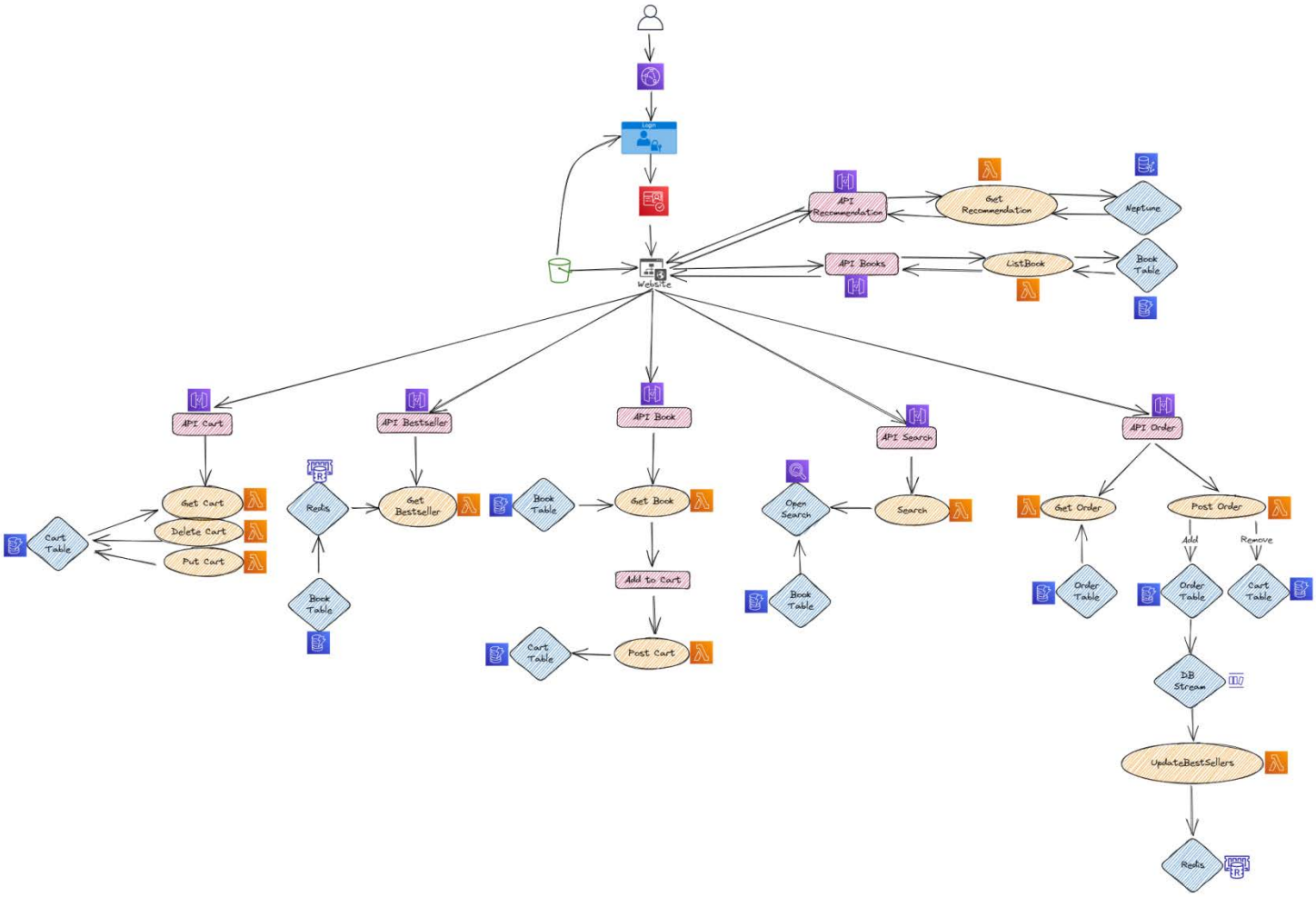
Amazon CloudWatch

The capabilities provided by CloudWatch are not exposed to the end users of the web app, rather the developer/administrator can use CloudWatch logs, alarms, and graphs to track the usage and performance of their web application.

AWS CodeCommit, AWS CodePipeline, AWS CodeBuild

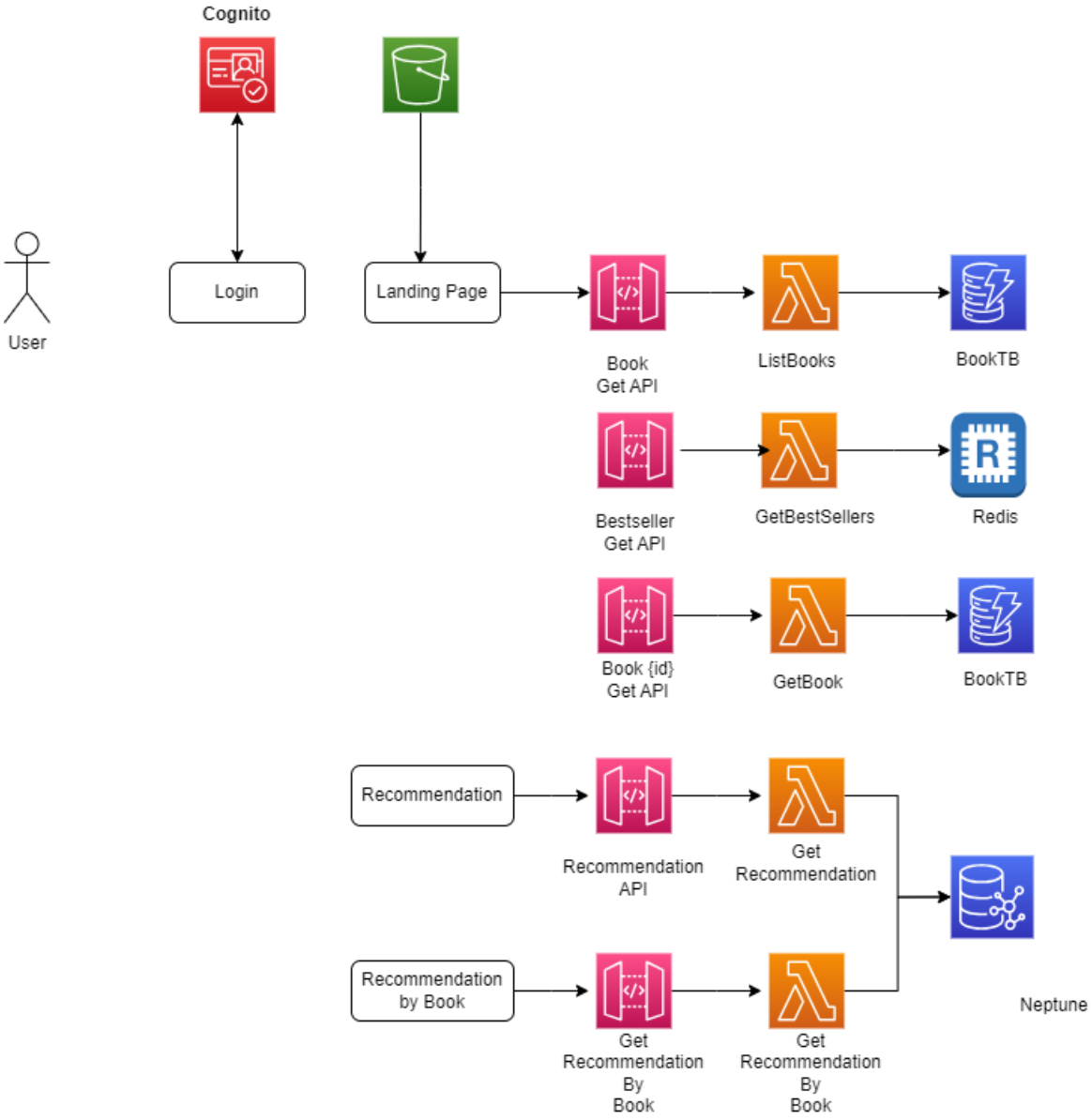
Similar to CloudWatch, the capabilities provided by CodeCommit, CodePipeline, and CodeBuild are not exposed to the end users of the web app. The developer/administrator can use these tools to help stage and deploy the application as it is updated and improved.

Flow Diagram (User perspective)



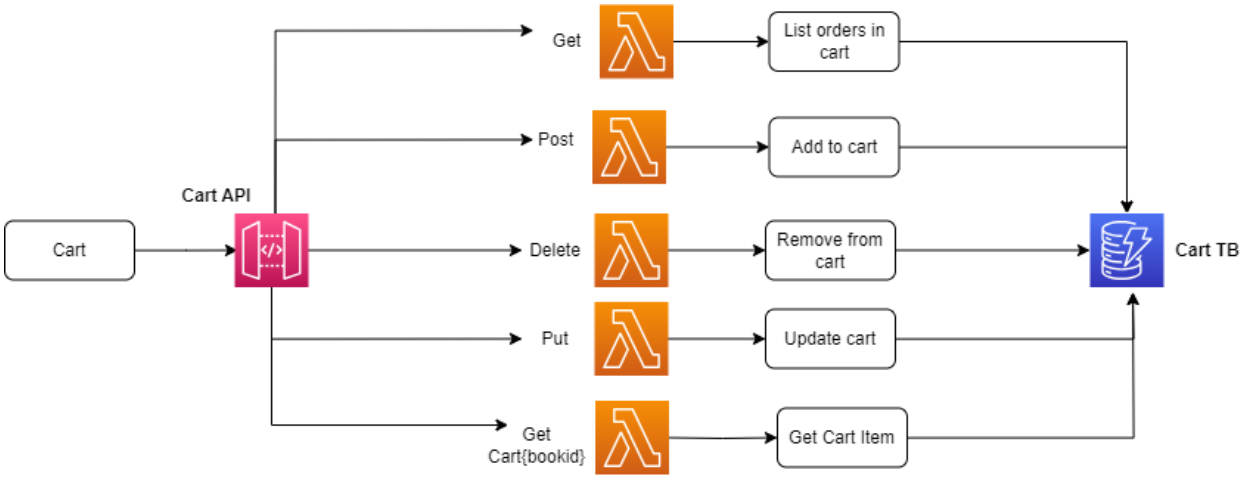
Flow Diagrams (Lambda Function Perspective)

1. **When the landing page loads.**

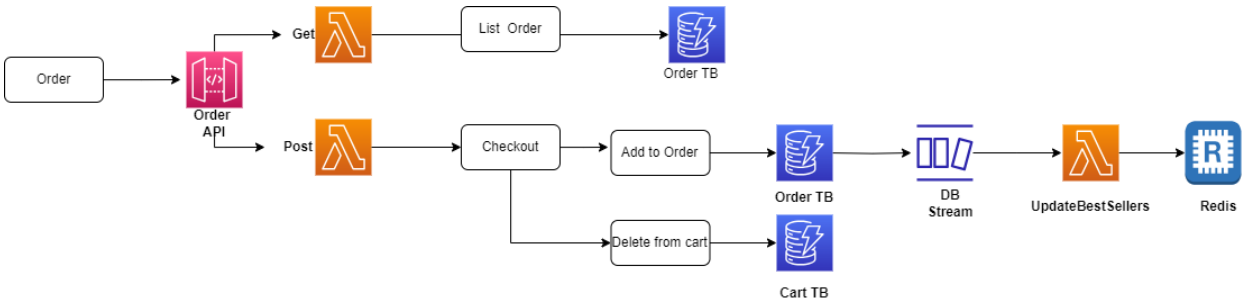


2. **Search for any particular book**

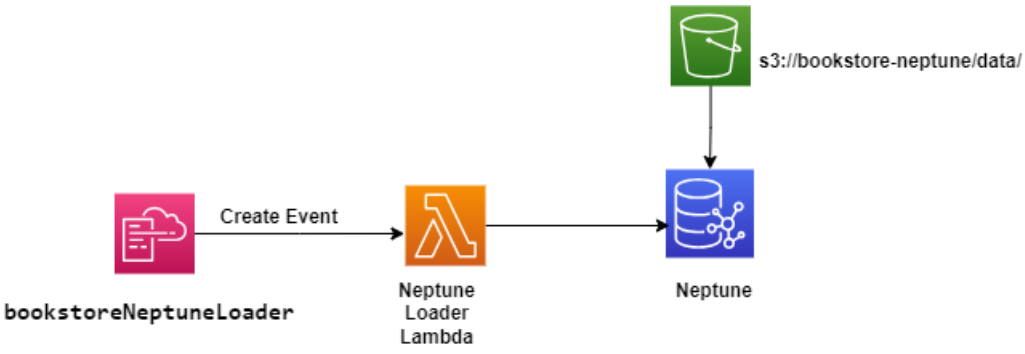
3. Cart related actions



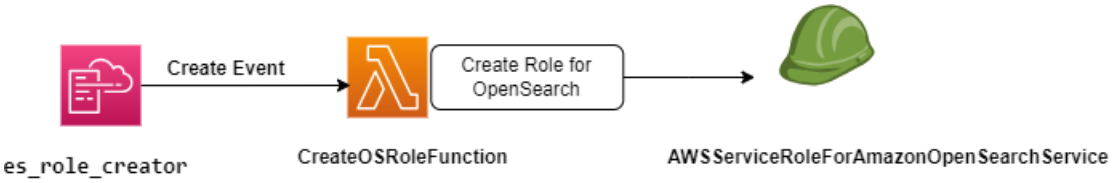
4. Order related actions



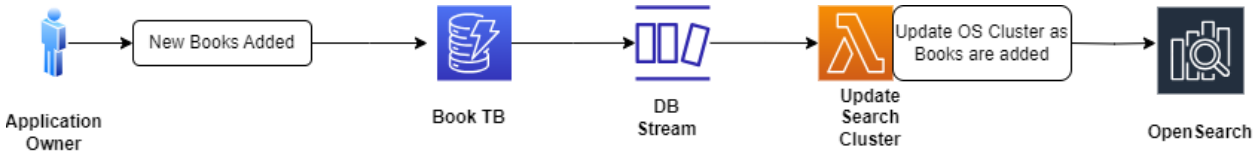
5. Loading data to Neptune



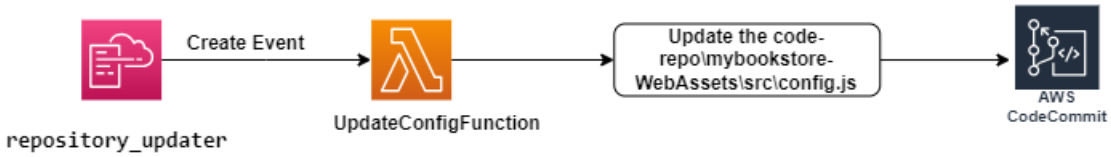
6. Creating OpenSearch Role



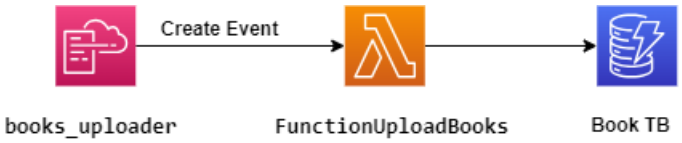
7. **Update the OpenSearch Cluster once the books are added or edited.**



8. **Updating the config code files available on CodeCommit Repo.**



9. **Books upload in DynamoDB**



Proposed Plan:

1. **Inventory**
2. **Warehouse Database**
3. **Admin console**