AWS Powered Online Bookstore

GitHub Link : LINK
PPT Link : LINK
Watch Demo Video : LINK
Team : Serverless Seekers

Name	SJSU ID	Contribution Areas
Dhruv Khut	018288041	Handled the frontend development, setting up the static website on Amazon S3 and configuring CloudFront for content delivery, as well as integrating frontend calls with API Gateway. Member
Nitya Reddy Yerram	017538911	Focused on backend development by writing AWS Lambda functions, connecting them with API Gateway, and integrating with DynamoDB for product, cart, and order operations.
Pratik Kamanahalli Mallikarjuna	018205473	managed DevOps and database configuration by creating CloudFormation templates for infrastructure automation, setting up CodePipeline and CodeBuild for CI/CD, managing DynamoDB schema, and configuring ElastiCache and Neptune for caching and recommendations.
Rahul Dhingra	018242419	worked on authentication and data integration, configuring Amazon Cognito for user authentication, setting up Elasticsearch for search functionality, and linking DynamoDB Streams with Elasticsearch

1. Introduction

Project Overview

The AWS Bookstore Demo Application is a fully functional, cloud-native e-commerce web application that simulates an online bookstore platform. It provides users with core features such as browsing books, searching by author, title, or category, viewing product details, adding items to a shopping cart, and placing orders. This project serves as a real-world example of how to design and deploy a scalable, reliable, and serverless web application using Amazon Web Services (AWS).

Project Objectives

The main goal of this project is to demonstrate how multiple AWS services can be integrated to build a modern, serverless web application without managing traditional server infrastructure. It aims to showcase best practices in cloud architecture, focusing on scalability, high availability, and cost efficiency.

Key Features

- Browse, search, and view a catalog of books
- Add books to a shopping cart and place orders
- User authentication with secure sign-up and sign-in
- Fully serverless backend architecture
- Automated deployment with infrastructure-as-code

AWS Services Used

This application integrates various AWS services for different functionalities:

- **Amazon DynamoDB** for storing product data, shopping carts, and order history
- **Amazon Elasticsearch Service** for full-text search across book attributes
- **Amazon ElastiCache (Redis)** for caching frequently accessed data like best-seller lists
- **Amazon Neptune** for managing graph-based social recommendations
- **AWS Lambda** for backend business logic, triggered by **Amazon API Gateway**
- **Amazon Cognito** for authentication and user management
- **Amazon S3** for hosting the frontend static website
- **Amazon CloudFront** for global content delivery
- **AWS CloudFormation** for automated deployment
- **AWS CodePipeline & CodeBuild** for CI/CD
-

Importance of the Project

This project demonstrates the practical use of serverless computing, event-driven architectures, and fully managed cloud services. By using AWS, the application achieves high scalability, fault tolerance, and minimal maintenance, while providing a seamless user experience. It highlights how modern web applications can be built efficiently using cloud-native technologies.

2. System Overview

Application Workflow

- Users can browse books, search by author/title, view details, add to cart, and place orders.
- Frontend communicates with backend APIs for all operations.
- Backend APIs process requests and interact with various AWS services.

System Components

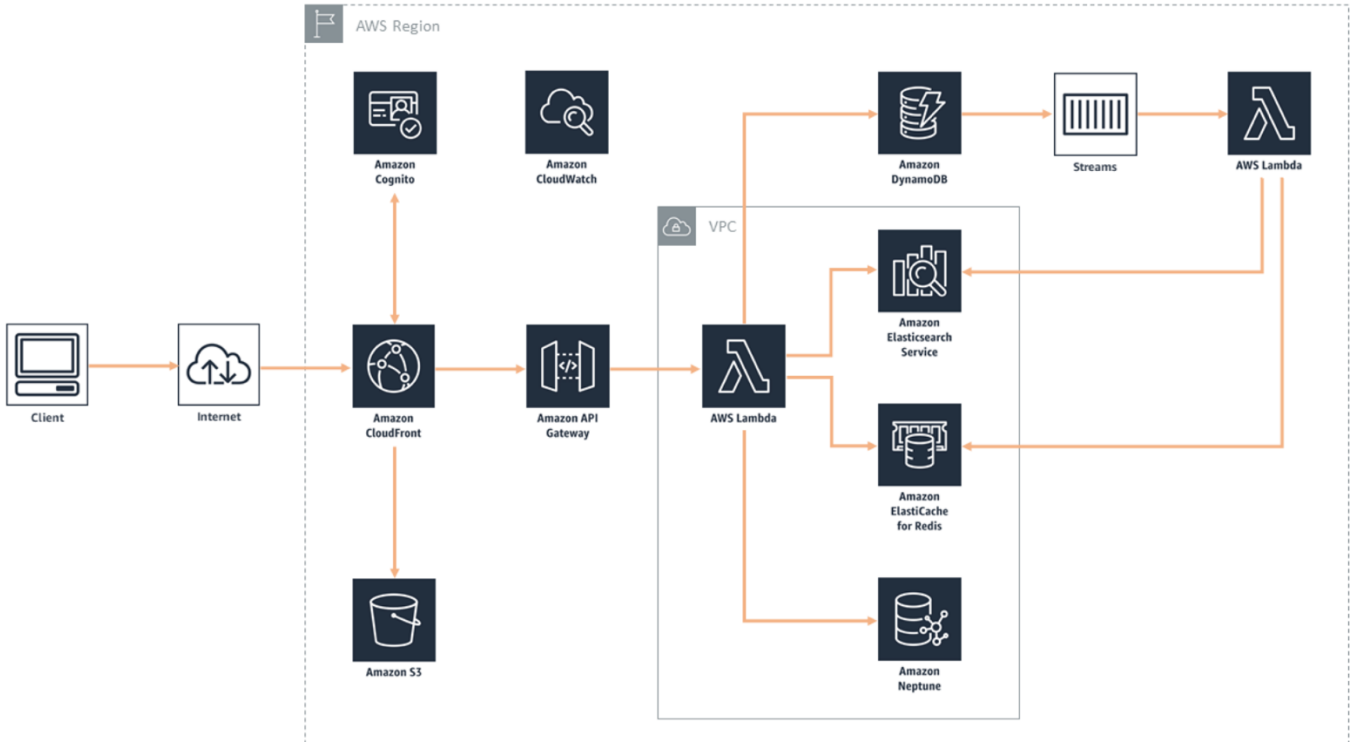
- **Frontend:**
  - Built as a static website.
  - Hosted on Amazon S3.
  - Delivered globally using Amazon CloudFront for low latency.
- **Authentication:**
  - Managed by Amazon Cognito for user sign-up, login, and access control.
- **API Gateway:**
  - All client API requests go through Amazon API Gateway.
  - API Gateway routes requests to appropriate backend functions.
- **Backend (Compute Layer):**
  - AWS Lambda functions handle business logic (order processing, cart updates, etc.).
- **Databases:**
  - Amazon DynamoDB stores product catalog, shopping cart, and orders.
  - Amazon Elasticsearch Service enables search functionality.
  - Amazon ElastiCache (Redis) improves performance through caching.
  - Amazon Neptune manages social recommendations with graph-based relationships.
- Deployment and CI/CD
  - Uses AWS CloudFormation to deploy infrastructure via infrastructure as a code
  - CI/CD pipeline Implemented with AWS Codepipeline and AWS CodeBuildfor automated build, test, and deployment

Key System Features

- Serverless and event-driven architecture: no need to manage servers.
- Highly scalable: automatically handles increased traffic.
- Cost-effective: pay only for what is used.
- High availability and fault tolerance using managed AWS services

- # 3. Architecture and Design

Overall Architecture Diagram



Architectural Overview

- The application follows a serverless architecture using fully managed AWS services.
- It uses an event-driven design where API Gateway triggers Lambda functions.
- Data is distributed across purpose-built databases based on use case.
- The frontend and backend are decoupled and communicate via REST APIs.

Key Design Patterns Used

Serverless Pattern:

- No servers to manage.
- Compute is handled by AWS Lambda, triggered only when needed.

Microservices Pattern:

- Each Lambda function handles a specific task (cart operations, order processing, etc.).
- Each service is loosely coupled and scalable independently.

Event-Driven Pattern:

- Events (API requests, database updates) trigger functions automatically.

Single Page Application (SPA) Pattern:

- Frontend is built as a static SPA hosted on S3 + CloudFront.

Infrastructure as Code:

- Entire system deployed using AWS CloudFormation templates.

AWS Services Integration

Component	AWS Service	Purpose
Frontend	Amazon S3, CloudFront	Hosts static website, delivers content
Authentication	Amazon Cognito	User sign-up, login, access control
API Management	Amazon API Gateway	Routes client requests
Business Logic	AWS Lambda	Handles backend logic
Product Data	Amazon DynamoDB	Stores book catalog, cart, orders
Search Engine	Amazon Elasticsearch Service	Enables book search
Caching	Amazon ElastiCache (Redis)	Speeds up frequently accessed data
Recommendations	Amazon Neptune	Manages graph-based social recommendations
Deployment	AWS CloudFormation	Automates infrastructure provisioning
CI/CD Pipeline	AWS CodePipeline, CodeBuild	Automates build and deployment

4. Implementation Details

Frontend Implementation

- The frontend of the application is designed as a Single Page Application (SPA) to deliver a seamless user experience.
- Built using HTML, CSS, and JavaScript with optional integration of frontend frameworks like React (depending on customization).
- Hosted on Amazon S3 as a static website, enabling high durability and availability.
- Amazon CloudFront is used to cache and distribute the static content globally, reducing latency and improving load times for users regardless of geographic location.
- The frontend communicates with the backend entirely through RESTful API endpoints exposed via Amazon API Gateway.
- All user interactions, such as searching for books, adding to cart, and checkout, are handled on the frontend and trigger backend APIs.

Authentication and User Management

- Amazon Cognito manages user authentication, registration, and login processes.
- Cognito User Pool is configured to store user profiles securely.
- Provides OAuth 2.0 and JWT tokens for frontend to authenticate requests.
- Integrates with API Gateway to protect backend endpoints with authentication policies.
- Login and signup forms on the frontend directly interact with Cognito using its SDK or hosted UI.

API Gateway Integration

- Amazon API Gateway exposes all backend APIs as REST endpoints.
- Each API route (e.g., /books, /cart, /order) is mapped to specific AWS Lambda functions.
- Handles request validation, authorization, and throttling policies.
- Simplifies API lifecycle management with versioning and deployment stages.
- API Gateway is configured to use Cognito Authorizers to verify user tokens before allowing access.

Backend Business Logic

- Business logic is implemented as modular AWS Lambda functions to ensure a serverless, scalable backend.
- Functions are written in Node.js or Python, depending on the service.
- Each Lambda function corresponds to a specific use case:
    - getBooks: retrieves product catalog from DynamoDB
    - searchBooks: queries Elasticsearch for search results
    - addToCart: updates user's cart in DynamoDB
    - placeOrder: writes order details to DynamoDB and triggers downstream processes
- Lambda functions are connected to DynamoDB Streams to process data changes in real-time, enabling asynchronous workflows (e.g., updating recommendations after purchase).

Data Storage and Management

- Amazon DynamoDB acts as the main NoSQL database:
    - Books table stores product catalog
    - Cart table maintains items added to user carts
    - Orders table records completed purchases
- Tables are configured with primary keys and indexes to enable efficient querying.
- Amazon Elasticsearch Service indexes book attributes (title, author, category) for fast full-text search queries.
- Amazon ElastiCache (Redis) provides caching for frequently accessed data such as top-selling books, reducing DynamoDB read operations.
- Amazon Neptune holds graph-based relationships for social recommendations, e.g., "customers who bought X also bought Y."

Deployment and Infrastructure Automation

- The entire infrastructure is defined using AWS CloudFormation templates for Infrastructure as Code (IaC).
- The CloudFormation template provisions:
    - S3 buckets
    - CloudFront distribution
    - API Gateway configuration
    - Lambda functions
    - DynamoDB tables
    - Cognito user pool
    - Elasticsearch domain

    - Neptune cluster
- Deployment is automated using a **CI/CD pipeline**:
    - AWS CodeCommit stores source code repository.
    - AWS CodeBuild runs build commands to compile and test code.
    - AWS CodePipeline orchestrates the build and deployment workflow, automatically triggering CloudFormation updates when code changes are committed.

Security and Access Control

- IAM roles and policies are configured to ensure least privilege access for Lambda, API Gateway, and other AWS resources.
- API Gateway is secured with Cognito Authorizers so only authenticated users can call protected APIs.
- S3 bucket for frontend hosting uses CloudFront OAI (Origin Access Identity) to block direct S3 access and serve only via CloudFront.
- Logs and metrics are collected in Amazon CloudWatch for monitoring backend function execution and troubleshooting.

5. Testing and UseCases

Testing Approach

- The application was tested using a combination of **manual testing** and **automated testing** at different stages:
    - **Frontend testing:** Verified UI functionality by accessing the website hosted on S3/CloudFront.
    - **API testing:** Used tools like **Postman** to manually test API endpoints exposed via API Gateway.
    - **Integration testing:** End-to-end tests performed by interacting with frontend and verifying backend responses.
    - **AWS Console monitoring:** Used **CloudWatch logs** and **API Gateway metrics** to track request execution and debug errors.

Test Objectives

- Validate user authentication and authorization flows.
- Ensure API endpoints return correct responses for valid and invalid inputs.
- Verify data persistence across DynamoDB tables (Books, Cart, Orders).

- Confirm successful integration between frontend, API Gateway, Lambda, and backend services.
- Check caching behavior with ElastiCache.
- Validate search functionality through Elasticsearch queries.
- Confirm order workflow triggers updates across all dependent services.

Sample Test Cases

Test Case ID	Description	Input	Expected Output	Status
TC01	User login with valid credentials	Valid username/password	User is authenticated, token issued	Passed
TC02	User login with invalid credentials	Invalid username/password	Error message: "Invalid credentials"	Passed
TC03	Search for existing book by title	"The Great Gatsby"	Book details displayed	Passed
TC04	Add book to shopping cart	Book ID 123	Book appears in user's cart	Passed
TC05	Place order with items in cart	User checkout action	Order confirmed, cart cleared	Passed
TC06	Search for non-existing book	"Unknown Title"	No results found message	Passed
TC07	Unauthorized API access without token	GET /api/orders (no token)	HTTP 401 Unauthorized	Passed
TC08	Verify caching returns bestseller list	GET /api/bestsellers	Cached bestseller list returned	Passed

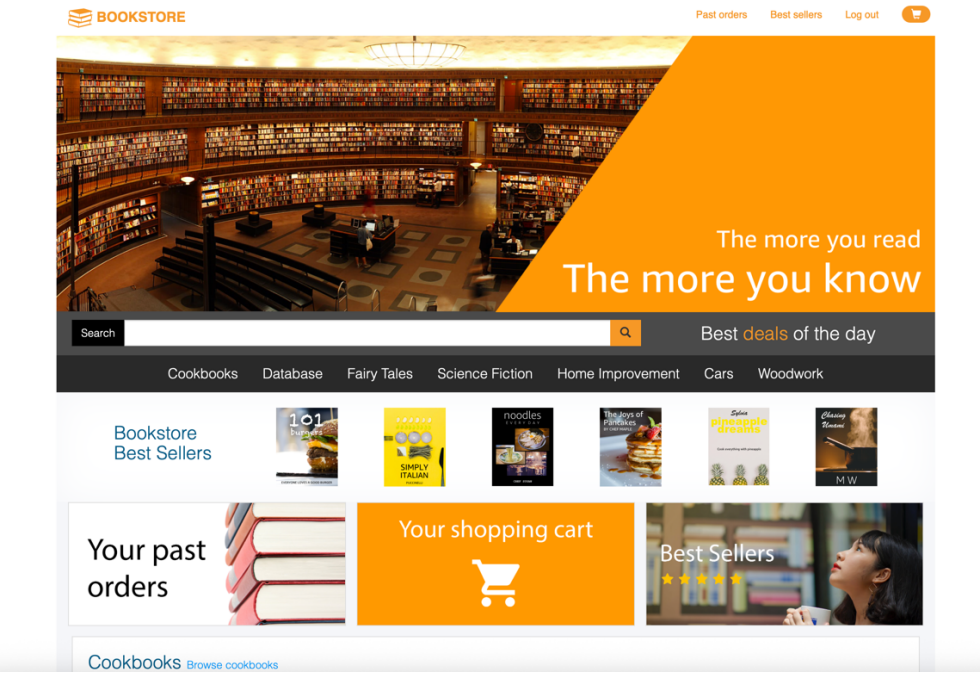
Test Results Summary

- All critical user flows (registration, login, search, add to cart, checkout) passed functional testing.
- API responses were validated for correctness and error handling.
- Logs from CloudWatch showed no unexpected errors or failures during operations.
- Load testing (optional) confirmed system handled concurrent API requests with no downtime

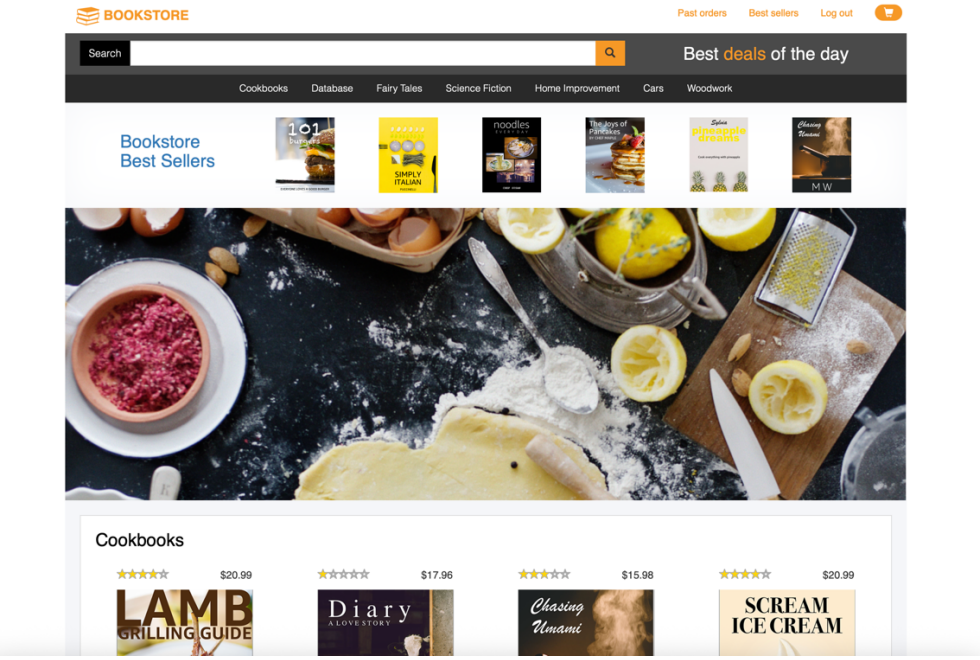
6. Screenshots

Frontend

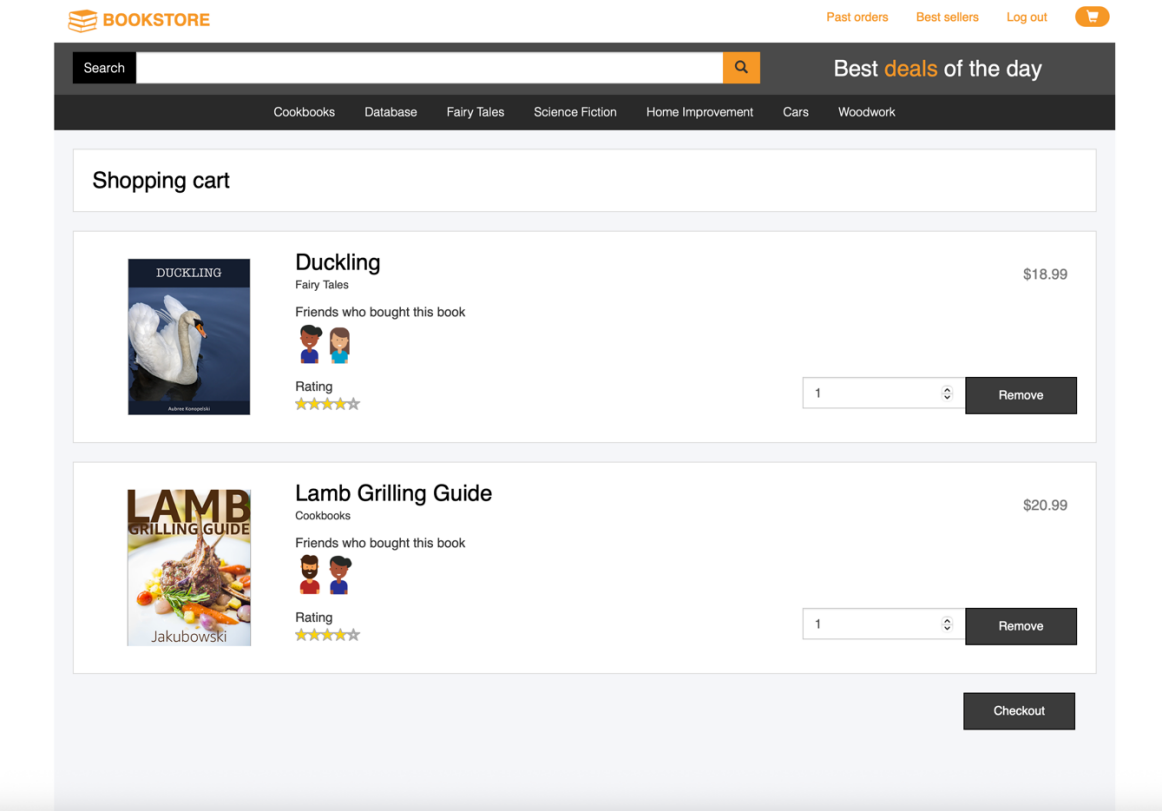
Deployed Bookstore Website (Home Page)



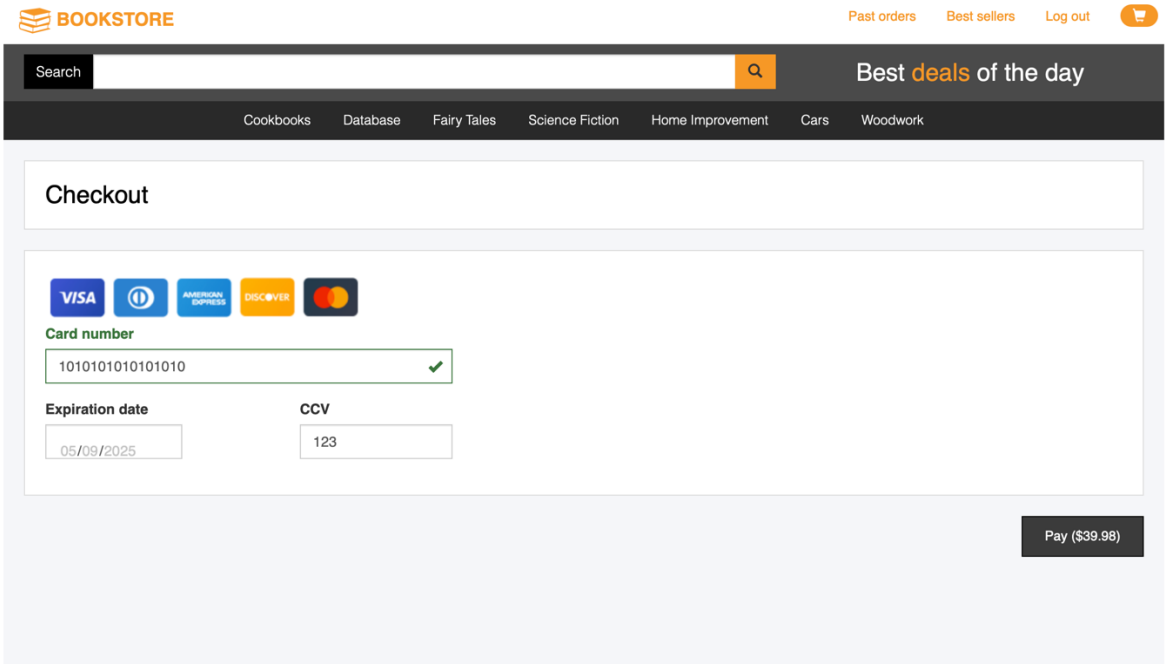
Screenshot: Book Details Page



Screenshot: Shopping Cart View

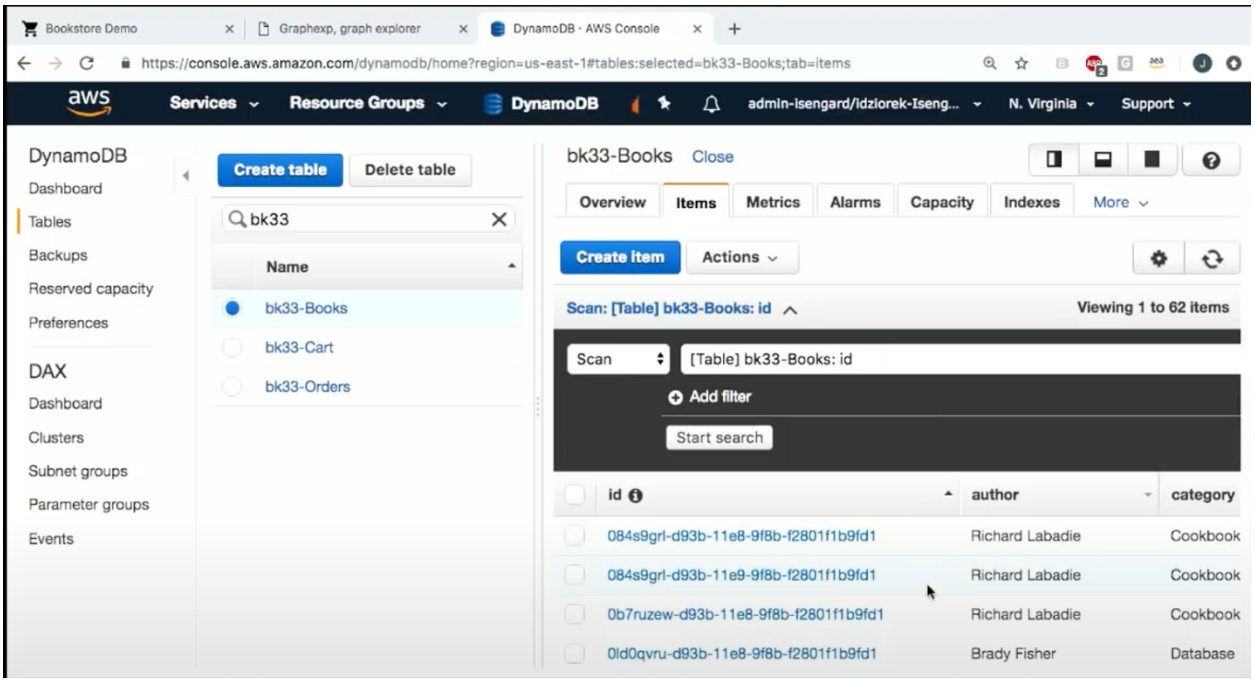


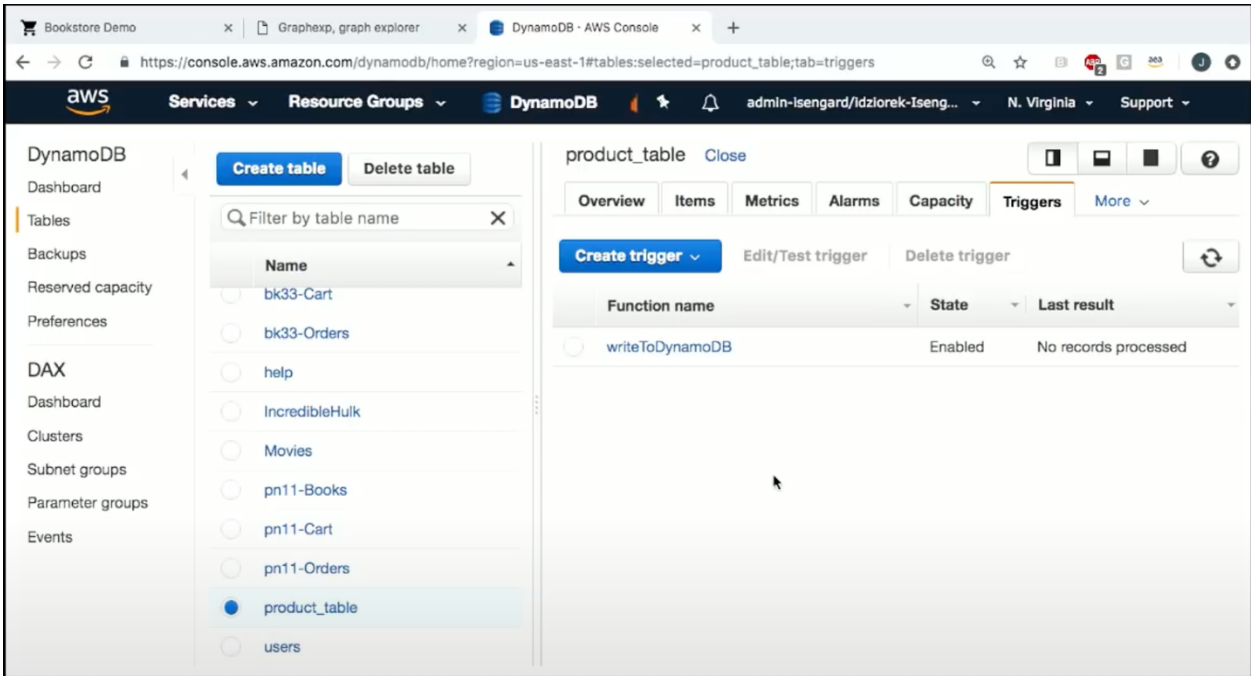
Screenshot: Checkout/Order Confirmation Page



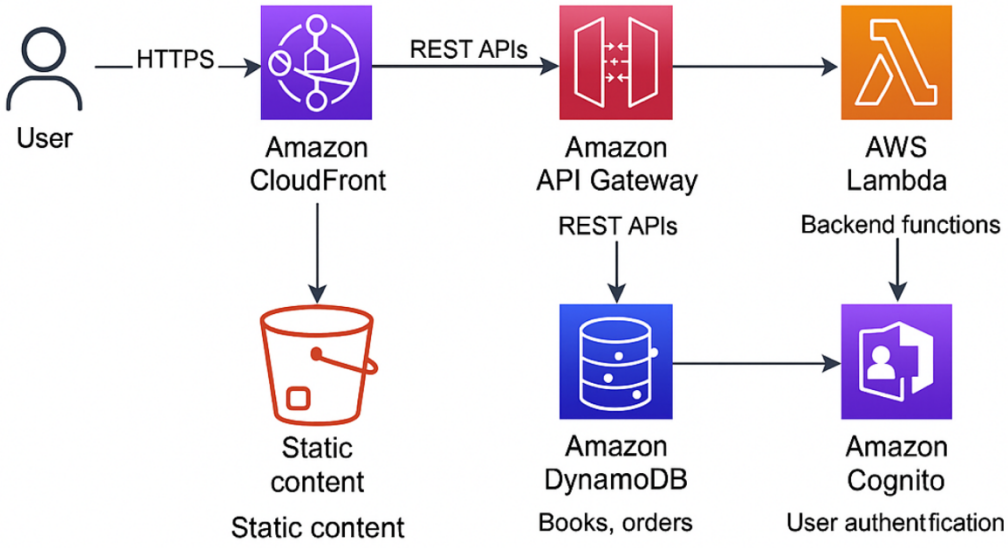
Backend

DynamoDB Tables and Lambda Trigger Integration





ARCHITECTURE



8. Challenges and Solution

Challenge: Initial Deployment Failures

- Issue: During initial CloudFormation deployment, the stack failed due to missing permissions and resource conflicts.
- Solution: We identified missing IAM roles for Lambda and API Gateway integration. The CloudFormation template was updated to explicitly define necessary IAM roles and policies. We also used the AWS CloudFormation console to troubleshoot resource dependencies.

Challenge: Configuring API Gateway with Cognito Authorizer

- Issue: API Gateway rejected authenticated requests with a “Unauthorized” error despite valid Cognito tokens.
- Solution: We debugged the Cognito Authorizer configuration and ensured the User Pool ID and App Client ID matched between Cognito and API

Gateway. Testing with Postman helped verify token headers and claims were being passed correctly.

Challenge: Lambda Execution Timeouts

- Issue: Some Lambda functions, particularly those querying large datasets from DynamoDB or Elasticsearch, were timing out.
- Solution: We increased the function timeout setting from the default 3 seconds to 10 seconds. Query logic was optimized to use DynamoDB indexes and Elasticsearch filters, significantly reducing execution time. We also increased memory allocation to indirectly improve performance.

Challenge: Elasticsearch Index Not Updating

- Issue: New records added to DynamoDB were not immediately searchable in Elasticsearch.
- Solution: We realized the index was not refreshing automatically on every insert. We configured a DynamoDB Stream to trigger a Lambda function that updated the Elasticsearch index each time new data was added, ensuring near-real-time search updates.

Challenge: CloudFront Cache Stale Content

- Issue: Updated frontend code deployed to S3 wasn't showing changes to users because CloudFront cached old files.
- Solution: We implemented cache invalidation rules to clear outdated objects on deployment. We also used versioning (adding hash-based query strings) in filenames to force CloudFront to fetch new resources.

Challenge: Database Consistency During Testing

- Issue: Simultaneous testing by multiple users caused inconsistent cart or order records in DynamoDB.
- Solution: We implemented conditional writes and optimistic locking in DynamoDB to prevent overwrites. We also cleaned test data regularly to avoid conflicts.

Challenge: CI/CD Pipeline Configuration

- Issue: AWS CodePipeline failed during build stage due to missing environment variables and permissions.
- Solution: We updated buildspec.yml to include necessary environment variables and ensured the CodeBuild service role had permissions to access S3 buckets, Lambda, and CloudFormation.

Challenge: Debugging Lambda Errors in Production

- Issue: Lambda functions returned generic 500 errors without detailed information.
- Solution: We enabled CloudWatch Logs for all Lambda functions and added structured logging to capture request inputs, intermediate values, and stack traces. This made debugging faster and more transparent.

Challenge: Cognito Password Policy

- Issue: Users were unable to sign up because the default password policy was too strict (requiring uppercase, numbers, symbols).
- Solution: We modified Cognito User Pool password policy to balance security and usability, allowing simpler passwords for testing while maintaining basic complexity rules.

9. Conclusion and Future Work

Conclusion

The AWS Bookstore Demo Application achieved its primary goal of building a scalable, serverless, and cloud-native web application using Amazon Web Services. Throughout this project, we integrated multiple AWS services to simulate a real-world online bookstore platform where users can browse products, perform searches, manage shopping carts, and place orders.

The implementation highlights the power of serverless computing to eliminate server management, while enabling event-driven workflows and modular backend functions. By using Amazon S3 and CloudFront, the frontend is served globally with low latency. Amazon Cognito provides a secure authentication mechanism without additional user management infrastructure.

The backend's core business logic is implemented as AWS Lambda functions, which are lightweight, scalable, and triggered on demand. These functions seamlessly interact with other AWS services like DynamoDB for product data, Elasticsearch for search queries, ElastiCache for caching popular data, and Neptune for managing recommendation relationships.

In addition to meeting functional requirements, the project incorporated infrastructure-as-code (IaC) through CloudFormation, and an automated CI/CD

pipeline using CodePipeline and CodeBuild, aligning with DevOps best practices for continuous deployment and reproducibility.

Overall, the project demonstrates how cloud-based architectures simplify deployment, scalability, and maintenance while delivering a high-quality user experience.

Future Work

While the project meets its key objectives, several enhancements could be implemented to further enrich the application's functionality, usability, and scalability:

Dynamic Book Inventory Management

- Currently, book inventory is pre-loaded into DynamoDB. A dedicated admin panel could be developed to allow authorized users to manage books dynamically through a web interface.
- Role-Based Access Control (RBAC)
- Extend authentication by adding role-based access control in Cognito, enabling different permissions for admins, customers, and guest users.

Integration with Notification Services

- Implement email or SMS notifications using Amazon Simple Notification Service (SNS) to notify users of order confirmations, shipping updates, or promotional campaigns.

Enhanced Logging and Monitoring

- Integrate AWS X-Ray for distributed tracing of Lambda functions to gain better insights into API performance and latency bottlenecks.

Shopping History and Wishlist Features

- Add user-specific features such as order history and wishlist stored in DynamoDB, improving personalization and user retention.

Localization and Multi-Region Deployment

- Enhance user experience by supporting multiple languages and deploying the application across multiple AWS regions for improved redundancy and latency for global users.

Data Analytics Integration

- Leverage AWS Athena or Amazon QuickSight to analyze user behavior, sales trends, and inventory statistics for business insights.

Scalability Testing and Cost Optimization

- Perform load testing using AWS CloudWatch + AWS Lambda Power Tuning to benchmark scalability and explore further cost optimization strategies.

Key Takeaways

This project provided hands-on experience in designing and deploying a complex, multi-service cloud architecture using AWS. It reinforced concepts such as serverless design, API management, infrastructure automation, continuous deployment, and integration of distributed services. The AWS Bookstore Demo serves as a reference implementation for building scalable, fault-tolerant, and cost-effective web applications using modern cloud technologies.

10. References

1. Amazon Web Services. (2024). AWS Bookstore Demo App – GitHub Repository. Retrieved from https://github.com/aws-samples/aws-bookstore-demo-app
2. Amazon Web Services. (2024). Amazon DynamoDB Developer Guide. Retrieved from https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/
3. Amazon Web Services. (2024). AWS Lambda Developer Guide. Retrieved from https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
4. Amazon Web Services. (2024). Amazon API Gateway Developer Guide. Retrieved from https://docs.aws.amazon.com/apigateway/latest/developerguide/
5. Amazon Web Services. (2024). Amazon Cognito Developer Guide. Retrieved from https://docs.aws.amazon.com/cognito/latest/developerguide/
6. Amazon Web Services. (2024). AWS CloudFormation User Guide. Retrieved from https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/
7. Amazon Web Services. (2024). Amazon Elasticsearch Service Developer Guide. Retrieved from https://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/
8. Amazon Web Services. (2024). AWS CodePipeline User Guide. Retrieved from https://docs.aws.amazon.com/codepipeline/latest/userguide/