Master of Applied Computer Science

(CSCI 5409: Advanced Cloud Computing – Summer2023)

Term Assignment Report

Instructor

Prof. Robert Hawkey

Submitted by: Margin Patel (B00918149)

1. What I built and what it is supposed to do?

Billventory is an Inventory management web-application designed to help businesses efficiently manage their stock levels, streamline billing processes, and receive notifications for low-stock items. The app provides four primary functionalities:

1. Manual Stock Entry: Users can manually input stock information into the app. When users receive new inventory, they can add details such as product name, quantity, price, and other relevant information through a user-friendly interface.

2. Automated Stock Entry: Billventory offers a convenient option for users to upload stock sheets. Users can prepare stock sheets in format of pdf containing product details and quantities. Upon uploading the stock sheet, the app will automatically parse the data and populate the inventory with the provided stock information. This feature saves time and reduces manual data entry errors.

3. Billing Functionality: When a customer makes a purchase, the user can create a bill through the app. The user will select the products sold and specify the quantities. The app will then automatically deduct the sold quantities from the inventory, keeping the stock levels accurate in real-time.

4. Low-Stock Notifications: To avoid stockouts, the app provides a notification system. When a product's quantity falls below a predefined threshold (set by the user), the app will trigger a low-stock alert. The user will receive an email notification, allowing them to take timely action by reordering the product.

Billventory's target audience includes businesses of various sizes that deal with inventory management and sales. By automating manual tasks, providing real-time updates, and offering active notifications, Billventory aims to optimize inventory control, reduce manual efforts, minimize stockouts, and enhance overall operational efficiency. The application's success will be evaluated based on its ability to streamline inventory management processes, deliver accurate billing, and effectively notify users about low-stock situations, leading to improved inventory handling and smoother business operations.

2. List of services and comparison:

- **Compute:**
    - **AWS EC2:** I have chosen EC2 (Elastic Compute Cloud) for hosting my application's frontend. It provides scalable virtual server instances to host my web application on the cloud. EC2 instances offer various configurations, and users have full control over the underlying infrastructure, making it suitable for custom applications like Billventory. EC2 provides the flexibility to choose the desired

operating system, install custom software, and configure security settings according to specific requirements.

- **AWS LAMBDA:** I have hosted all my backend services on AWS Lambda. It is a serverless compute service. It allows me to run code without provisioning or managing servers. In Billventory, AWS Lambda is also utilized for automated stock entry and low-stock notifications. These functions can be triggered automatically when specific events occur, such as stock sheet uploads or inventory updates. By using Lambda, Billventory eliminates the need to manage servers and ensures that these functions scale automatically without any additional administrative overhead.
- **Comparison with alternative Services:** Beanstalk may lack the customization needed for infrastructure [1]. It does not provide flexibility and control like EC2. Docker Containers are hard for lambda to run and required more expertise for coding. Implementing Step-funtions can be more challenging for backend.

- **Storage:**
  - **Amazon DynamoDB:** I have stored inventory details, users' details, and bills details in DynamoDB. It is a fully managed NoSQL database with fast and predictable performance, automatic scaling, and a flexible schema, suitable for the dynamic inventory data. It stores as key-value and is easy for data retrieval.
  - **S3:** I have stored the pdf files of stock sheet in S3 as it is highly durable and scalable object storage service, ideal for storing uploaded PDFs and static files.
  - **Comparison with alternative Services:** AWS RDS is relational database and so backend has to be depend on relational database. RDS is costly then DynamoDB [2].Neptune is for graph database so this is beyond scope of app. AWS Athena is for SQL querying and doing analysis. [3]
- **Network:**
  - **AWS API Gateway:** I have used AWS API Gateway for making REST API's for backend. Through it, frontend can securely connect to backends API and can invoke lambda.
  - **Comparison with alternative service:** AWS VPC require additional cost for setting of virtual private cloud [4]. API gateway is more quantifiable then VPC in respect of everything as it can be easily secure using API keys. Further IAM roles can be assigned to every other resources so that they cannot be accessible by others.
- **General:**
  - **AWS Textract:** For parsing stock sheets, I integrated AWS Textract. This machine learning service automatically extracts text and data from scanned documents, enabling me to populate the inventory with accurate information.

- **SNS (Simple Notification Service):** To send low-stock notifications via email to users, I implemented Amazon SNS. It is a fully managed notification service that offers the flexibility to distribute messages to various endpoints, including email.
- **Alternative:** While Amazon Comprehend is useful for analyzing and understanding text for natural language processing, it is not optimized for extracting structured data from documents like stock sheets. Textract's ability to accurately extract structured data made it the preferred choice.

3. Deployment Model :

- The Deployment model of my project Billventory is public cloud. There are few reasons why public cloud was selected and are mentioned below:
  - Cost-Effectiveness: Public cloud deployments eliminate the need for significant upfront capital expenditures on hardware and infrastructure. This cost-effective approach allows me to allocate resources efficiently and scale as the application grows.
  - Global Reach: By deploying Billventory on a public cloud, I can take advantage of the cloud provider's (AWS) global infrastructure, ensuring that my application is easily accessible to users worldwide.
  - Reliability: AWS offer high availability and redundancy, reducing the risk of downtime and ensuring reliable access to Billventory for users.
  - Scalability: The public cloud's elastic nature enables me to scale resources dynamically, controlling fluctuations in user demand and ensuring consistent performance.

  Public cloud is best choice as private clouds require significant upfront investments in hardware, software, and infrastructure, making them less cost-effective for a startup like Billventory with budget constraints. In addition, hybrid cloud deployments involve integrating and managing both on-premises infrastructure and public cloud services. This complexity might not be suitable for a startup with limited IT resources and expertise.

4. Delivery model :

- I have choose AWS's two delivery model – IaaS and FaaS. Choosing IaaS (Infrastructure as a Service) for the frontend of my Billventory application offers several advantages that make it a suitable choice:
  - Customization and Control: With IaaS, I have full control over the frontend infrastructure. I can customize the virtual machines, networking, and other resources to meet the specific requirements of my frontend application. This level of control allows me to fine-tune the performance, security, and configurations to best suit my needs.

    - <u>Resource Scaling:</u> Amazon EC2 (IaaS service) provide the ability to scale frontend resources vertically (resizing VMs) or horizontally (adding more VM instances) based on demand. This elasticity ensures that my frontend can handle varying levels of user traffic and maintain optimal performance during peak times. I currently uses EC2's tc2.medium instance for hosting frontend but if in future more traffic is increased I can easily switch over to t2.xLarge or can also add new t2.medium and scale it.
    - <u>Cost Control:</u> EC2 offer various pricing models, such as pay-as-you-go or reserved instances, enabling me to manage costs effectively.
- For backend services, I have used Function-as-a-Service (FaaS). Main benefits for choosing this model is given below:
    - <u>Event-Driven Architecture:</u> Lambda is designed for event-driven architectures, making it well-suited for handling specific backend tasks triggered by events, such as stock updates, low-stock notifications, and billing processes. Each backend function can be designed to respond to a particular event, reducing the complexity of managing the entire backend.
    - <u>Serverless Abstraction:</u> Lambda abstracts away the need to manage servers and infrastructure. With using Lambda I don't have to worry about server provisioning, scaling, or monitoring. This serverless model allows me to focus on writing code and implementing business logic without getting distracted by infrastructure management.
    - <u>Cost Efficiency:</u> With Lambda, I have to only pay for the actual execution time and resources used by each function. This granular billing model can lead to cost savings.
- Users uses my application as SaaS (Software as a Service). Users can access the application through a web browser without needing to install or manage any software locally. This approach eliminates the need for users to deal with any infrastructure or technical complexities, making it easy for them to start using Billventory immediately.

5. Final architecture:

- **Cloud mechanisms fit together**
    - I have hosted frontend on ec2 t2.medium and will get public access URL to use the web-app. The backend of the application is built using serverless functions with AWS Lambda.
    - User will signup first, this will hit the lambda for authentication through apigateway. This lambda will send subscriber mail to that user through Simple Notification Service(SNS). User will then redirected to home inventory page where all stocks are visible. At this page getproducts lambda will be hitted to fetch all the inventory details from DynamoDB.

- User can then navigate to Add stock section where he will enter stocks details manually. When submit is clicked , lambda is executed for adding products to inventory DynamoDB. In addition user can upload pdf of stocks. When pdf is uploaded, lambda is executed which upload pdf to S3 bucket and then automatically S3 triggers a PUT event to lambda which perform Textract to pdf and parse all data and further stores to DynamoDB.
- When user want to generate bill, he/she can enter basic info and can click button to generate. When clicked, lambda for generating bill is triggered, which will reduce quantity in inventory and update it. For viewing all the bills, there is lambda which fetch all the data from DynamoDB Bills. This lambda also check whether inventory has low quantity, If yes then it will send low quantity message through SNS to the login user.

- **Datastore**
  - For storing data I am using S3 and DynamoDB for as mentioned below purposes:
    - Simple Storage Service(S3): I am storing pdf of the stock sheets in S3 which can trigger event to lambda. S3 is best suited for this.
    - DynamoDB: I am storing user information of password and emailId in DynamoDB users table. Inventory details of productid, name etc are also stored in DynamoDB products table. Generated bills are also stored in bills table.
- **Programming Languages**
  - **ReactJs:** Frontend of whole project was done in React. React is best suited for single page application which can provide smooth interaction with users. MaterialUI provide best design for frontend which is only for React. React app was deploy in EC2 through cloudformation.
  - **Node.js:** Node.js is well-suited for serverless environments and integrates seamlessly with AWS Lambda, allowing me to write the backend logic as JavaScript functions and deploy them effortlessly. There are huge resources for Node.js to learn for lambda. All the backend logic requires coding and was written in Node.js.
  - **YAML:** YAML's simple and readable syntax makes it convenient for writing and maintaining configuration files. It is more easily readable then JSON format. I have used it for creating Cloudformation script which enables all resources.
  - **Shell Script:** I have used shell commands for deploying frontend to EC2. This command is written in cloudformation script only.
- **How is system deployed?**
  - For deployment, my whole project is deploy through one CloudFormation script only. In that script, parameters, resources and outputs are defined.
  - As soon as script is run, frontend React code is deployed on EC2. All Lambda functions are created with their respective codes. S3 bucket is also created for

storing pdf. Three DynamoDb tables are created. API-Gateway for all lambdas are initialize and URL of all the API is passed as AWS Systems Manager. URL for accessing app is given as output for cloudformation.

- **Architecture**
  - Currently I am not following any architecture. But, In future if demand increase for the application, then I can follow Workload Distribution Architecture. In this architecture, I can add two more EC2 instance to distribute the load. Load balancer can be configure for this which can distribute traffic equally.
  - I am not following Elastic Disk Provisioning Architecture as I am not using Elastic block storage (EBS). Currently, I want NoSQL database so using DynamoDB.

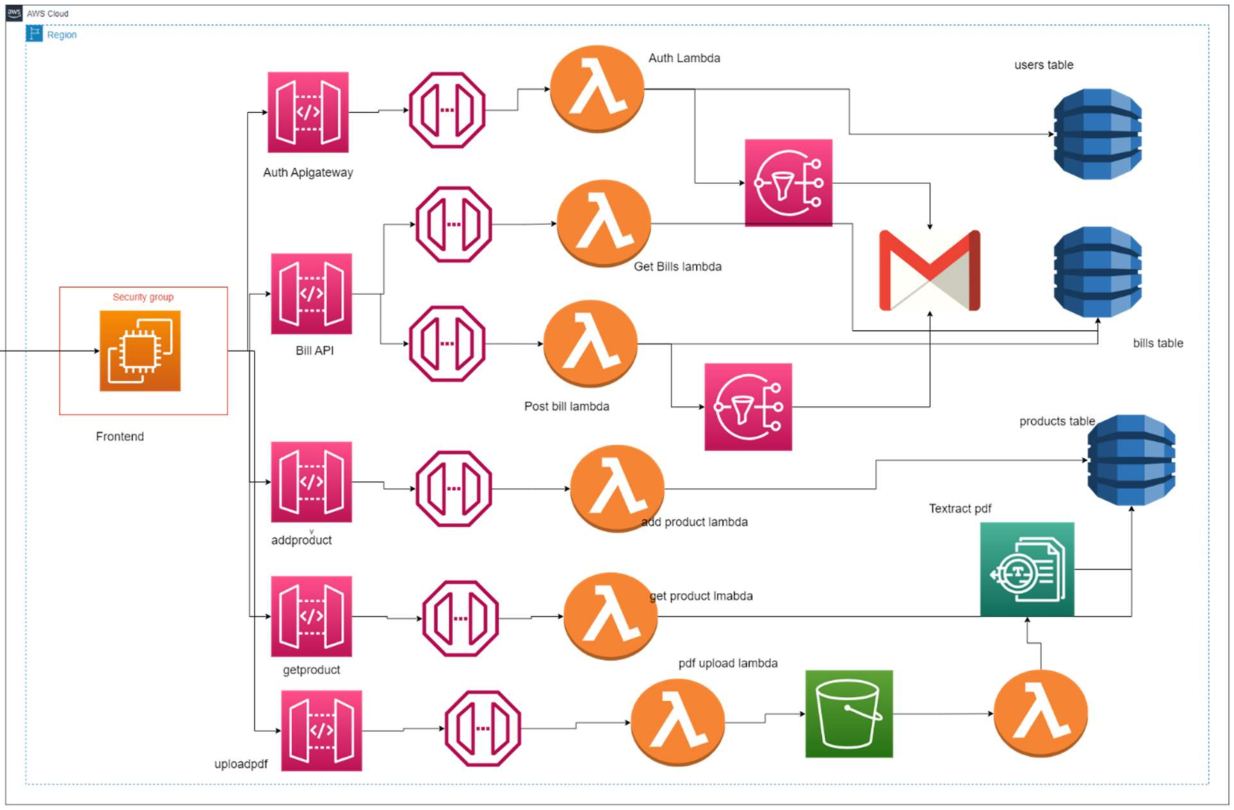

Fig 1. Final Architecture

6. Data security at all layers:

- **Computer Layer (EC2 & Lambda):** For securing the EC2 react app, I have created Security Group. This security group will only give access to port 80 and port 3000. Any other port are not accessible. For SSH, there is security key associated to it which is required everytime for SSH into EC2. Without SSH, no one can see the react code. For backend as code is running on lambda, AWS is authorize to look security for it.
- **Nework Layer (ApiGateway):** For securing the network of frontend-backend communication, APIGateway is used. It enforce the use of HTTPS for all API endpoints

to encrypt data in transit and ensure secure communication. So data in transit is encrypted by AWS valid SSL certificate.

- **Storage Layer (S3 & DynamoDB):** IAM roles and policies are defined for accessing S3 and DynamoDB. Only Lambda functions can access both of them.
- **Security Mechanisms:**
  - Hashing mechanism is done for passwords. All user passwords are encrypted and store in DynamoDB.
  - Files in S3 are server side encrypted so data at rest is encrypted. Same goes for DynamoDB.
  - AWS Systems Manager is used for accessing API-Gateways all URL in frontend. This SSM is protected by the Accesskey, secretkey and session-token, so it is not accessible by all.

7. Reproduce architecture in a private cloud:

- **Server & Networking cost:** The cost of purchasing servers will vary based on their specifications, including CPU, memory, and storage capacity. Average cost is between $5000 to $10000. Red Hat provides 2 CPU for $65000. VMware vRealizet take $8000 per CPU's. For similar project working on bigger scale, more than 4 servers are needed to handle load. So overall, $15000-$20000 are estimated. [5]

  For networking purpose switches, routers, firewalls, load balancers, and other networking components are required. For purchasing 1 load-balances average $10000 price is there [6]. 4 Routers and 6 switches of Cisco can cost upto $8000. [7]

- **Software Licenses cost:** For parsing the pdf files, AWS textract is to be used which can cost upto $100 per month. SSL certificates can cost upto $1000 per year. [8]
- **Storage cost:** For storing pupose 100Tb hard-drive and 10Tb SSD can cost upto $40000.
- **Internet cost:** Good internet connection can cost upto $500 per month.
- **Backup cost:** Additional storage for backup purpose can need 50TB HDD which can cost $10000.

8. Monitoring:

- Cloud mechanism which can incur high cost is EC2. My frontend is hosted on EC2 and I have to monitor it as it cost can escalates suddenly if it is kept running on continuously. As backend is on lambda and it is FaaS, I will pay-as-use. So, no need to monitor backend. I can keep budget alarm for ec2 so that in future it can notify me.

9. Evolve over time:

- In future if I got chance to carry on this project, I will add more features as mentioned below:
    - I. Authentication with Cognito : I will use AWS Cognito for authentication purpose. JWT token will be generated and user can safely use the app.
    - II. AWS QuickSight for analysis of Inventory: New dashboard will be created for analysis in which month which product is sold more. AWS Quicksight can help for this as it can generate graphs and diagrams.
    - III. AWS Lex for Chat Assistant: User interaction can be increased by integrating LEX with the frontend as it will talk to user and can give basic info. User can ask directly product name and Lex can fetch the quantity from DynamoDB.

References

[1] "Beanstalk," AWS, [Online]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html. [Accessed 07 2023].

[2] "RDS vs DynamoDb," AWS, [Online]. Available: https://cloudacademy.com/blog/amazon-rds-vs-dynamodb-12-differences/. [Accessed 2023].

[3] "AWS Athena," AWS, [Online]. Available: https://aws.amazon.com/athena/. [Accessed 07 2023].

[4] "VPC pricing," AWS, [Online]. Available: https://aws.amazon.com/vpc/pricing/. [Accessed 07 2023].

[5] "Private cloud pricing," clinked, [Online]. Available: https://blog.clinked.com/private-cloud-pricing. [Accessed 2023].

[6] "Price of load balancer," Load Balancer org, [Online]. Available: https://www.loadbalancer.org/. [Accessed 2023].

[7] "Routers & Switches," router-switch, [Online]. Available: https://www.router-switch.com/Price-cisco-switches_c2. [Accessed 07 2023].

[8] "textract pricing," AWS, [Online]. Available: https://aws.amazon.com/textract/pricing/. [Accessed 2023].